Programmer un logiciel multicorps en 2D

Franck RENAUD et Jean-Luc DION

20 octobre 2024

Table des matières

1 Solide et repères

1.1 Rotation et position

Soit un solide S :

- Soit le point O, origine du repère Galiléen $(\underline{X}_0, \underline{Y}_0, \underline{Z}_0)$
- Soit le point G, centre de gravité du solide S et origine du repère $(\underline{X}_S, \underline{Y}_S, \underline{Z}_S)$ attaché au solide S
- Soit le point P élément du solide S et origine du repère $(\underline{X}_L, \underline{Y}_L, \underline{Z}_L)$ attaché au solide S

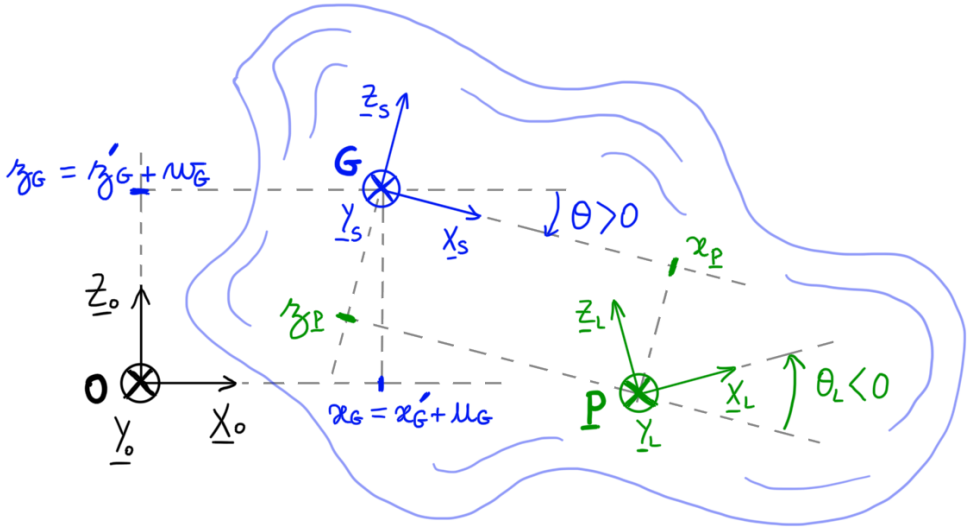

FIGURE 1 – *Paramétrage d'un solide en 2D*

Les degrés de liberté du solide S sont concaténés dans le vecteur $\underline{q}_S$. Les deux premiers degrés de libertés (ddl) $x_S = x_G$ et $z_S = z_G$ correspondent à la position du centre de gravité dans le repère Galiléen. Le troisième ddl $\theta_S = \theta$ est l'orientation angulaire du solide.

$$\underline{q}_S = \left\{ \begin{array}{c} x_S \\ z_S \\ \theta_S \end{array} \right\} = \left\{ \begin{array}{c} x_G \\ z_G \\ \theta \end{array} \right\}$$

Exprimons les vecteurs du repère du solide S dans le repère galiléen :

$$\left\{ \begin{array}{l} \underline{X}_S = \cos\theta \ \underline{X}_0 - \sin\theta \ \underline{Z}_0 \\ \underline{Y}_S = \underline{Y}_0 \\ \underline{Z}_S = \sin\theta \ \underline{X}_0 + \cos\theta \ \underline{Z}_0 \end{array} \right.$$

Ainsi, on a :

$$\underline{X}_S^{\mathscr{B}_0} = \left\{ \begin{array}{r} \cos\theta \\ -\sin\theta \end{array} \right\} \begin{array}{l} \rightarrow \underline{X}_0 \\ \rightarrow \underline{Z}_0 \end{array} \qquad \text{et} \qquad \underline{Z}_S^{\mathscr{B}_0} = \left\{ \begin{array}{r} \sin\theta \\ \cos\theta \end{array} \right\} \begin{array}{l} \rightarrow \underline{X}_0 \\ \rightarrow \underline{Z}_0 \end{array}$$

Soit $\underline{\underline{R}}_{0\leftarrow S}$ la matrice qui permet de passer de la base $\mathscr{B}_S$ du solide à la base $\mathscr{B}_0$ du repère galiléen. C'est également la matrice de rotation du solide S :

$$\underline{\underline{R}}_{0\leftarrow S} = \left[\left\{ \underline{X}_S^{\mathscr{B}_0} \right\} , \left\{ \underline{Z}_S^{\mathscr{B}_0} \right\} \right] = \left[\begin{array}{cc} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{array} \right] \tag{1}$$

Le solide S étant considéré indéformable, le point P est immobile dans le repère du solide, alors le vecteur $\underline{GP}^{\mathscr{R}_S}$ est constant. La position du point P dans le repère galiléen peut être exprimée de la manière suivante :

$$\underline{OP}^{\mathscr{R}_0} = \underline{OG}^{\mathscr{R}_0} + \underline{GP}^{\mathscr{R}_0} = \underline{OG}^{\mathscr{R}_0} + \underline{\underline{R}}_{0\leftarrow S}\underline{GP}^{\mathscr{R}_S} \tag{2}$$

Code Python 1.
Créez un fichier Python nommé **utils2D.py**
Programmez les fonctions `rotation` et `position`.

```
import numpy as np


def rotation(q):
    """
    Parameters
    ----------
    q : numpy array (3, )

    Returns
    -------
    R : numpy array (3, 3)
    """
    theta = q[2]
    c = np.cos(theta)
    s = np.sin(theta)
    R = np.array([[c, s], [-s, c]])
    return R


def position(q, GP_Rs):
    """
    Parameters
    ----------
    q     : numpy array (3, )
    GP_Rs : numpy array (2, )

    Returns
    -------
    P : numpy array (2, )
    """
    R = rotation(q)
    P = q[0:2] + R @ GP_Rs
    return P
```

1.2 Architecture du code Python

Comme montré en Figure 1, chaque solide disposera d'un ou plusieurs repères de liaison $\mathscr{R}_L$ (en vert), positionnés et orientés par rapport au repère du solide lui-même $\mathscr{R}_S$ (en bleu). Le repère du solide sera quant à lui positionné et orienté par rapport au repère galiléen $\mathscr{R}_0$ (en noir).

Afin de modéliser un mécanisme, il est pratique de s'appuyer sur des points caractéristiques tels que les centres de liaison. Souvent, nous connaissons la position de ces points par rapport au repère galiléen, $\underline{OP}^{\mathscr{R}_0}$. Il s'agit alors de les exprimer par rapport au repère d'un solide, $\underline{GP}^{\mathscr{R}_S}$. Pour cela nous pouvons inverser l'équation (2) :

$$\underline{GP}^{\mathscr{R}_S} = \left(\underline{\underline{R}}_{0\leftarrow S}\right)^{-1}\left(\underline{OP}^{\mathscr{R}_0} - \underline{OG}^{\mathscr{R}_0}\right) = \underline{\underline{R}}^{t}_{0\leftarrow S}\,\underline{GP}^{\mathscr{R}_0} \tag{3}$$

Il nous manque l'angle $\theta_{L/S}$ du repère de liaison par rapport au repère du solide. On a : $\theta_{L/S} = \theta_{L/0} - \theta_{S/0}$. L'angle $\theta_{L/0}$ du repère de liaison par rapport au repère galiléen peut être simplement déduit de la connaissance du vecteur $\underline{X}_L^{\mathscr{R}_0} = \alpha\underline{X}_0 + \beta\underline{Z}_0$ exprimé dans le repère galiléen. Il est en effet plus pratique pour l'utilisateur de définir cet axe afin qu'il corresponde à une réalité mécanique (l'axe d'une glissière par exemple). On a alors $\theta_{L/S} = \arctan(-\beta/\alpha) - \theta_{S/0}$

Nous allons maintenant définir une architecture de code Python qui nous permette de décrire un solide et les repères qui lui sont associés.

- La classe `Solid` comportera les propriétés du solide : nom, masse, inertie, position $\underline{q}_0$ et vitesse $\dot{\underline{q}}_0$ initiale du repère de solide. Elle comportera également un dictionnaire des points particuliers appartenant au solide : repères de liaisons et points de mesure pour le post-traitement. Chacun de ces points sera défini par une classe `Frame`.
- La classe `Frame` contiendra toutes les propriétés d'un repère : son nom (label), sa position dans un repère parent et son orientation par rapport au repère parent.

Code Python 2.
Créez un fichier Python nommé **Multicorps2D.py**
Programmez la classe `Frame`.

```
import numpy as np
from utils2D import rotation, position


class Frame:
    def __init__(self, label="frame",
                 P=np.zeros(2), theta=0):
        self.label = label
        self.position = P
        self.theta = theta
```

Les méthodes associées à la classe `Solid` sont :

- `add_frame` qui ajoutera un repère de classe `Frame` au dictionnaire des repères
- `plot_t0` qui tracera grossièrement la position des repères d'un solide à l'aide de `matplotlib`. Afin de distinguer chaque solide dans une même figure, nous utiliserons une couleur et un type de marque distincts par solide. Pour cela les propriétés `color` et `marker` seront ajoutés à cette classe.

Code Python 3.
A la suite du fichier Python **Multicorps2D.py**
Programmez la classe `Solid`, en remplissant les `?_____?` par le code qui convient :

```
class Solid:
    def __init__(self, label="solid", m=1.0, iy=0.01,
                 G_t0=np.zeros(2), theta_t0=0,
                 dG_t0=np.zeros(2), dtheta_t0=0,
                 color=[0, 0, 1], marker='+'):
        self.label = label
        self.m = m
        self.iy = iy
        self.ind_q = []  # Indices des ddls de position
        self.ind_dq = [] # Indices des ddls de vitesse
        self.q_t0 = np.concatenate((G_t0, [theta_t0]))
        self.dq_t0 = np.concatenate(?___?)
        self.frames = {"G": Frame(label="G")}
        self.color = color
        self.marker = marker

    def add_frame(self, tag_F="frame", label="frame",
                  P_t0=np.zeros(2), Xs=np.array([1., 0.])):
        R = rotation(self.q_t0)
        P_Rs = R.T @ (P_t0 - self.q_t0[0:2])
        theta = np.arctan2(-Xs[1], Xs[0]) - self.q_t0[2]
        self.frames[tag_F] = Frame(?___?)

    def draw_t0(self, ax):
        Nb = len(self.frames)
        G = self.q_t0[0:2]
        P = np.zeros((2*Nb, 2))
        ind = 0
        for item, value in self.frames.items():
            P[ind, :] = G
            P[ind+1, :] = position(self.q_t0, value.position)
            ind += 2
        ax.plot(P[:, 0], P[:, 1], label=self.label,
                color=self.color, marker=self.marker,
                fillstyle='none', markersize=8)
```

Afin de tester le code précédent, nous allons contruire quelques solides constitués de plusieurs repères et afficher le résultat dans une figure Matplotlib.

Exercice 1.
Créez un fichier Python nommé **Main_cours.py** qui appelle les classes et les fonctions de **Multicorps2D.py**

Correction de l'exercice 1.
Créez un fichier Python nommé **Main_cours.py** qui contient :

```
import numpy as np
import Multicorps2D as mc
import matplotlib.pyplot as plt

# Definition of points in the galilean frame
pts = {}
pts['G1'] = np.array([1, 1])
pts['G2'] = np.array([1, 2])
pts['G3'] = np.array([2, 2])
pts['G4'] = np.array([2, 1])
pts['A'] = np.array([0, 0])
pts['B'] = np.array([2, 3])
pts['C'] = np.array([0, 4])
pts['D'] = np.array([4, 0.5])
pts['E'] = np.array([0, 2])
pts['F'] = np.array([2.5, 0])

# Creation of solid 1
s1 = mc.Solid(label="Carter", G_t0=pts['G1'])
s1.add_frame(tag_F="A", label="Axe pivot 1", P_t0=pts['A'])
s1.add_frame(tag_F="B", label="Axe pivot 2", P_t0=pts['B'])
s1.add_frame(tag_F="C", label="Rotule 1", P_t0=pts['C'])
s1.add_frame(tag_F="D", label="Capteur", P_t0=pts['D'])

# Creation of solid 2
s2 = mc.Solid(label="Bielle", G_t0=pts['G2'],
              color=[0, 0.7, 0], marker='o')
s2.add_frame(tag_F="A", label="A", P_t0=pts['A'])
s2.add_frame(tag_F="C", label="C", P_t0=pts['C'])
s2.add_frame(tag_F="E", label="E", P_t0=pts['E'])
s2.add_frame(tag_F="F", label="F", P_t0=pts['F'])

# Creation of solid 2
s3 = mc.Solid(label="Roue dentee", G_t0=pts['G3'],
              color=[0.75, 0.75, 0], marker='s')
s3.add_frame(tag_F="D", label="D", P_t0=pts['D'])
s3.add_frame(tag_F="G2", label="G2", P_t0=pts['G2'])
s3.add_frame(tag_F="G4", label="G4", P_t0=pts['G4'])

# Plotting the solids
fig, ax = plt.subplots()
ax.grid()
s1.draw_t0(ax)
s2.draw_t0(ax)
s3.draw_t0(ax)
ax.legend()
```

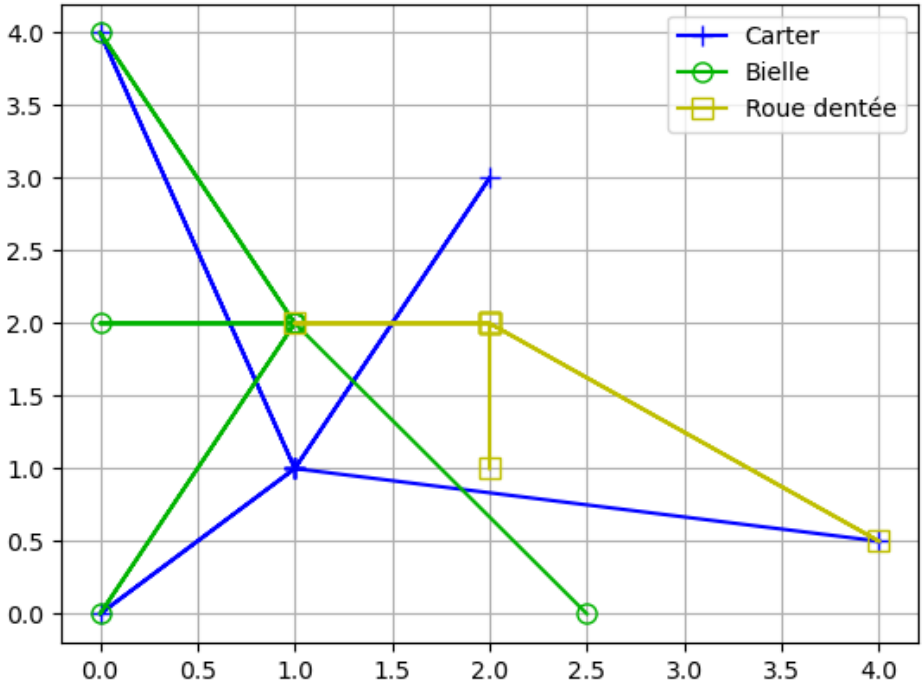

FIGURE 2 – *Affichage des solides dans leur état initial*

1.3 Inspection des variables Python

Si l'on cherche à afficher la variable `s1` du code précédent, on obtiendra :

Console Python 1.

```
>>> s1
<Multicorps2D.Solid at 0x274aa705d90>

>>> s1.frames['A']
<Multicorps2D.Frame at 0x274aa705dd0>
```

Afin de pouvoir vérifier ce que contiennent nos variables, il est nécessaire de modifier la méthode `__repr__` des classes `Frame` et `Solid`.

Code Python 4.
Ajoutez à la suite de la définition de la classe `Frame`, la méthode suivante :

```
def __repr__(self):
    s  = "'{}'\n".format(self.label)
    temp = "\tx = {:+.3f} m, \tz = {:+.3f} m, " +\
        "\ttheta = {:+.3f} rad\n"
    s += temp.format(self.position[0], self.position[1],
                     self.theta)
    return s
```

Ajoutez à la suite de la définition de la classe `Solid`, la méthode suivante :

```
    def __repr__(self):
        s  = "'{}' :\n".format(self.label)
        temp = "\tx_0 = {:+.3f} m, \tz_0 = {:+.3f} m, " + \
            "\tttheta_0 = {:+.3f} rad\n"
        s += temp.format(self.q_t0[0], self.q_t0[1], self.q_t0[2])
        s  += "\tm = {:.3f} kg, \tiy = {:.3f} kg.m^2\n" \
            .format(self.m, self.iy)
        for key, value in self.frames.items():
            s += "\t.frames[{}] = ".format(key)
            s += value.__repr__().replace("\n\t", "\n\t\t")
        return s
```

Le code précédent donnera :

Console Python 2.

```
>>> s1
'Carter' :
    x_0 = +1.000 m,     z_0 = +1.000 m,     theta_0 = +0.000 rad
    m = 1.000 kg,    iy = 0.010 kg.m^2
    .frames[G] = 'G'
            x = +0.000 m,    z = +0.000 m,    theta = +0.000 rad
    .frames[A] = 'Axe pivot 1'
            x = -1.000 m,    z = -1.000 m,    theta = -0.000 rad
    .frames[B] = 'Axe pivot 2'
            x = +1.000 m,    z = +2.000 m,    theta = -0.000 rad
    .frames[C] = 'Rotule 1'
            x = -1.000 m,    z = +3.000 m,    theta = -0.000 rad
    .frames[D] = 'Capteur'
            x = +3.000 m,    z = -0.500 m,    theta = -0.000 rad
```

1.4 Vitesse et accélération

En repartant de l'équation (2), la vitesse du point P dans le repère galiléen peut être exprimée de la manière suivante :

$$\underline{\dot{OP}}^{\mathscr{R}_0} = \underline{\dot{OG}}^{\mathscr{R}_0} + \underline{\dot{GP}}^{\mathscr{R}_0} = \underline{\dot{OG}}^{\mathscr{R}_0} + \underline{\underline{\dot{R}}}_{0\leftarrow S}\underline{GP}^{\mathscr{R}_S} \tag{4}$$

Avec :

$$\underline{\underline{\dot{R}}}_{0\leftarrow S} = \dot{\theta}\begin{bmatrix} -\sin\theta & \cos\theta \\ -\cos\theta & -\sin\theta \end{bmatrix} = \dot{\theta}\underline{\underline{R}}_v \tag{5}$$

On peut ainsi écrire la vitesse du point P sous forme matricielle

$$\underline{\dot{OP}}^{\mathscr{R}_0} = \underline{\underline{T}}_v(S,P)\ \underline{\dot{q}}_S \qquad \text{avec} \qquad \underline{\underline{T}}_v(S,P) = \left[\underline{\underline{I}}_{2\times 2}\ ,\ \underline{\underline{R}}_v(\theta)\ \underline{GP}^{\mathscr{R}_S}\right] \tag{6}$$

Code Python 5.
A la suite du fichier Python **utils2D.py**, programmez les méthodes `rotation_v` qui rend $\underline{\underline{R}}_v$, `transformation_v` qui rend $\underline{\underline{T}}_v$ et `velocity` qui rend $\underline{\dot{OP}}^{\mathscr{R}_0}$:

```
    def rotation_v(q):
        """
        Parameters
        ----------
        q : numpy array (3, )

        Returns
        -------
        R_v : numpy array (3, 3)
        """
        ?___?

    def transformation_v(q, GP_Rs):
        """
        Parameters
        ----------
        q     : numpy array (3, )
        GP_Rs : numpy array (2, )

        Returns
        -------
        T_v : numpy array (2, 3)
        """
        Id2 = np.eye(2)
        temp = (rotation_v(q) @ GP_Rs)[:, np.newaxis]
        T_v = np.concatenate([ Id2, temp ], axis=1 )
        return T_v

    def velocity(q, dq, GP_Rs):
        """
        Parameters
        ----------
        q     : numpy array (3, )
        dq    : numpy array (3, )
        GP_Rs : numpy array (2, )

        Returns
        -------
        dP : numpy array (2, )
        """
        ?___?
```

L'accélération du point P dans le repère galiléen peut être exprimée de la manière suivante :

$$\underline{\ddot{OP}}^{\mathscr{R}_0} = \underline{\ddot{OG}}^{\mathscr{R}_0} + \underline{\ddot{GP}}^{\mathscr{R}_0} = \underline{\ddot{OG}}^{\mathscr{R}_0} + \underline{\underline{\ddot{R}}}_{0\leftarrow S}\underline{GP}^{\mathscr{R}_S} \tag{7}$$

Avec :

$$\underline{\underline{\ddot{R}}}_{0\leftarrow S} = \ddot{\theta}\begin{bmatrix} -\sin\theta & \cos\theta \\ -\cos\theta & -\sin\theta \end{bmatrix} + \dot{\theta}^2\begin{bmatrix} -\cos\theta & -\sin\theta \\ \sin\theta & -\cos\theta \end{bmatrix} = \ddot{\theta}\underline{\underline{R}}_v + \dot{\theta}^2\underline{\underline{R}}_a \tag{8}$$

On peut ainsi écrire l'accélération du point P sous forme matricielle :

$$\underline{\ddot{OP}}^{\mathscr{R}_0} = \underline{\underline{T}}_v(S,P)\ \underline{\ddot{q}}_S + \underline{\underline{T}}_a(S,P)\ \underline{\dot{q}}_S \qquad \text{avec} \qquad \underline{\underline{T}}_a(S,P) = \left[\underline{\underline{0}}_{2\times 2}\ ,\ \dot{\theta}\ \underline{\underline{R}}_a(\theta)\ \underline{GP}^{\mathscr{R}_S}\right] \tag{9}$$

Code Python 6.
A la suite du fichier Python **utils2D.py**, programmez les méthodes `rotation_a` qui rend $\underline{\underline{R}}_a$, `transformation_a` qui rend $\underline{\underline{T}}_a$ et `acceleration` qui rend $\underline{\ddot{OP}}^{\mathscr{R}_0}$:

```
def rotation_a(q):
    ?___?


def transformation_a(q, dq, GP_Rs):
    ?___?


def acceleration(q, dq, d2q, GP_Rs):
    ?___?
```

Afin de vérifier que vos fonctions ont bien été programmées voici quelques exemples de résultats que vous pouvez comparer aux votres dans la console Python de Spyder :

Console Python 3.

```
>>> from utils2D import *
>>> q = np.array([1, 0, np.pi/2])
>>> dq = np.array([0, 1, 1])
>>> d2q = np.array([1, 0, 1])
>>> GP_Rs = np.array([10, 10])

>>> print(rotation(q))
[[ 6.123234e-17  1.000000e+00]
 [-1.000000e+00  6.123234e-17]]

>>> print(position(q, GP_Rs))
[ 11. -10.]

>>> print(rotation_v(q))
[[-1.000000e+00  6.123234e-17]
 [-6.123234e-17 -1.000000e+00]]

>>> print(transformation_v(q, GP_Rs))
```

```
[[  1.   0. -10.]
 [  0.   1. -10.]]

>>> print(velocity(q, dq, GP_Rs))
[-10.  -9.]

>>> print(rotation_a(q))
[[-6.123234e-17 -1.000000e+00]
 [ 1.000000e+00 -6.123234e-17]]

>>> print(transformation_a(q, dq, GP_Rs))
[[  0.   0. -10.]
 [  0.   0.  10.]]

>>> print(acceleration(q, dq, d2q, GP_Rs))
[-1.90000000e+01 -1.77635684e-15]
```

2 Torseurs cinématiques

Le torseur cinématique du solide S, exprimé en son centre de gravité G vaut :

$$\left\{\mathscr{V}_S^{\mathscr{R}_0}\right\}_G = \left\{\underline{\Omega}_{S/\mathscr{R}_0} \mid \underline{V}_{G\in S/\mathscr{R}_0}\right\} = \left\{\dot{\theta}\underline{Y}_0 \mid \underline{\dot{OG}}^{\mathscr{R}_0}\right\} = \left\{\dot{\theta}\underline{Y}_0 \mid \dot{x}\underline{X}_0 + \dot{z}\underline{Z}_0\right\}$$

Le torseur cinématique du solide S, exprimé en un point P quelconque vaut :

$$\left\{\mathscr{V}_S^{\mathscr{R}_0}\right\}_P = \left\{\underline{\Omega}_{S/\mathscr{R}_0} \mid \underline{V}_{P\in S/\mathscr{R}_0}\right\} = \left\{\underline{\Omega}_{S/\mathscr{R}_0} \mid \underline{V}_{G\in S/\mathscr{R}_0} + \underline{\Omega}_{S/\mathscr{R}_0} \wedge \underline{GP}\right\}$$

Or nous venons de voir une façon différente d'exprimer la vitesse du point P. Prenons le temps de vérifier que les deux approches sont équivalentes :

$$\begin{array}{rcccl} \underline{V}_{P\in S/\mathscr{R}_0} & = & \underline{\dot{OG}}^{\mathscr{R}_0} + \dot{\underline{\underline{R}}}_{0\leftarrow S}\, \underline{GP}^{\mathscr{R}_S} & \overset{?}{=} & \underline{V}_{G\in S/\mathscr{R}_0} + \underline{\Omega}_{S/\mathscr{R}_0} \wedge \underline{GP} \\ & & \underline{\dot{OG}}^{\mathscr{R}_0} + \dot{\underline{\underline{R}}}_{0\leftarrow S}\, \underline{\underline{R}}^t_{0\leftarrow S}\, \underline{GP}^{\mathscr{R}_0} & \overset{?}{=} & \underline{V}_{G\in S/\mathscr{R}_0} + \underline{\Omega}^{\mathscr{R}_0}_{S/\mathscr{R}_0} \wedge \underline{GP}^{\mathscr{R}_0} \end{array}$$

La vitesse $\underline{\dot{OG}}^{\mathscr{R}_0}$ du point G apparait dans les deux cas, donc la question se résume à l'égalité des quantités suivantes :

$$\forall \underline{\nu} \in \mathbb{R}^2, \text{ as-t-on} \qquad \dot{\underline{\underline{R}}}_{0\leftarrow S}\, \underline{\underline{R}}^t_{0\leftarrow S}\, \underline{\nu} \;=\; \underline{\Omega}^{\mathscr{R}_0}_{S/\mathscr{R}_0} \wedge \underline{\nu} \qquad ?$$

Tout d'abord, remarquons que les matrices de rotation sont orthonormales et donc :

$$\underline{\underline{R}}\, \underline{\underline{R}}^t = \underline{\underline{I}} \quad \Longrightarrow \quad \frac{d}{dt}\left(\underline{\underline{R}}\, \underline{\underline{R}}^t\right) = \underline{\underline{0}} \quad \Longrightarrow \quad \dot{\underline{\underline{R}}}\, \underline{\underline{R}}^t = -\underline{\underline{R}}\, \dot{\underline{\underline{R}}}^t = -\left(\dot{\underline{\underline{R}}}\, \underline{\underline{R}}^t\right)^t$$

Puisque $\dot{\underline{\underline{R}}}\, \underline{\underline{R}}^t$ est égale à l'opposée de sa tranposée, c'est donc une matrice anti-symétrique. Or les matrice anti-symétriques décrivent des opérations de produit vectoriel. On a ici :

$$\dot{\underline{\underline{R}}}_{0\leftarrow S}\, \underline{\underline{R}}^t_{0\leftarrow S}\, \underline{\nu} = \dot{\theta} \begin{bmatrix} -\sin\theta & \cos\theta \\ -\cos\theta & -\sin\theta \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \underline{\nu} = \dot{\theta} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \underline{\nu}$$

Remarquons ensuite que $\underline{\Omega}^{\mathscr{R}_0}_{S/\mathscr{R}_0} = \dot{\theta}\, \underline{Y}_0$ et que le produit vectoriel suivant $\underline{Y}_0$ conduit à :

$$\begin{cases} \underline{Y}_0 \wedge \underline{X}_0 = -\underline{Z}_0 \\ \underline{Y}_0 \wedge \underline{Z}_0 = \quad \underline{X}_0 \end{cases} \qquad \Longrightarrow \qquad \underline{Y}_0 \wedge \underline{\nu} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \underline{\nu}$$

Donc :

$$\underline{\Omega}_{S/\mathscr{R}_0} \wedge \underline{\nu} = \dot{\theta} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \underline{\nu} \qquad \text{CQFD}$$

Les deux formules de vitesse sont donc équivalentes d'un point de vue mathématiques. En revanche, d'un point de vue programmation, il est préférable de tout réexprimer sous forme matricielle plutôt que de garder le produit vectoriel et il est préférable de garder le vecteur $\underline{GP}^{\mathscr{R}_S}$ qui a le mérite d'être constant.

Ainsi, on utilisera les torseurs cinématiques réel et virtuel suivants :

$$\left\{\mathscr{V}_S^{\mathscr{R}_0}\right\}_P = \left\{\dot{\theta}\underline{Y}_0 \mid \dot{x}\underline{X}_0 + \dot{z}\underline{Z}_0 + \dot{\theta}\underline{\underline{R}}_v(\theta)\underline{GP}^{\mathscr{R}_S}\right\} \tag{10}$$

$$\left\{\mathscr{V}_S^{\mathscr{R}_0}\right\}^*_P = \left\{\dot{\theta}^*\underline{Y}_0 \mid \dot{x}^*\underline{X}_0 + \dot{z}^*\underline{Z}_0 + \dot{\theta}^*\underline{\underline{R}}_v(\theta)\underline{GP}^{\mathscr{R}_S}\right\} \tag{11}$$

3 Energie cinétique

L'énergie cinétique de l'ensemble d'un système est la somme des énergies cinétiques de ses solides :

$$T = \sum_k T_k \quad \text{avec} \quad T_k = \frac{1}{2} m \underline{V}^2_{G \in S/\mathscr{R}_0} + \frac{1}{2} \underline{\Omega}^t_{S/\mathscr{R}_0} \underline{\underline{I}}_G \underline{\Omega}_{S/\mathscr{R}_0} = \frac{1}{2} m \left(\dot{x}^2 + \dot{z}^2 \right) + \frac{1}{2} I_y \dot{\theta}^2 \tag{12}$$

En effet, on a :

$$\underline{\Omega}^t_{S/\mathscr{R}_0} \underline{\underline{I}}_G \underline{\Omega}_{S/\mathscr{R}_0} = \langle 0,\ \dot{\theta},\ 0 \rangle \begin{bmatrix} ? & ? & ? \\ ? & I_y & ? \\ ? & ? & ? \end{bmatrix} \begin{Bmatrix} 0 \\ \dot{\theta} \\ 0 \end{Bmatrix} = I_y \dot{\theta}^2$$

4 Puissance virtuelle des accélérations $\mathscr{A}^*$

La méthode de Lagrange donne :

$$\mathscr{A}^* = \sum_{k=1}^{Nddl} \left(\frac{d}{dt} \frac{\partial T_k}{\partial \dot{q}_k} - \frac{\partial T_k}{\partial q_k} \right) \dot{q}^*_k \tag{13}$$

Dans le cas d'un unique solide $\mathscr{A}^* = A_x \dot{x}^* + A_z \dot{z}^* + A_\theta \dot{\theta}^*$

$$A_x = m\ \ddot{x}\ , \qquad A_z = m\ \ddot{z} \qquad \text{et} \qquad A_\theta = I_y\ \ddot{\theta}$$

Ainsi, la puissance virtuelle des accélérations d'un unique solide vaut :

$$\mathscr{A}^* = \left(\underline{\dot{q}}^*_S \right)^t \underline{\underline{M}}_S\ \underline{\ddot{q}}_S \qquad \text{avec} \qquad \underline{\underline{M}}_S = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & I_y \end{bmatrix} \tag{14}$$

Dans le cas de plusieurs solides la matrice de masse complète est obtenue en concaténant sur sa diagonale toutes les matrices de masses $\underline{\underline{M}}_S$ de chaque solide, cf. section 7.

Code Python 7.
Dans la classe `Solid`, ajoutez, la méthode `mass` qui renvoie $\underline{\underline{M}}_S$.

```
def mass(self):
    """
    Returns
    -------
    M : numpy array (3, 3)
    """
    ?___?
```

5 Torseurs des efforts

Le torseur d'un effort extérieur appliqué au point P du solide S et exprimé au même point P vaut :

$$\{\mathscr{F}_{ext\to P\in S}\}_P = \{\underline{F}_{ext\to S} \mid \underline{M}_{ext\to P\in S}\}$$

Le même torseur exprimé au point A vaut :

$$\{\mathscr{F}_{ext\to P\in S}\}_A = \{\underline{F}_{ext\to S} \mid \underline{M}_{ext\to P\in S} + \underline{F}_{ext\to S} \wedge \underline{PA}\}$$

5.1 Pesanteur

L'effort dû à la gravité est présent chez tous les solides, son torseur exprimé au centre de gravité vaut :

$$\{\mathscr{F}_{poids\to G\in S}\}_G = \{-mg\underline{Z}_0 \mid \underline{0}\}$$

5.2 Ressort et amortisseur entre 2 solides

Soient un ressort et un amortisseur reliant le point A du solide S_A au point B du solide S_B.

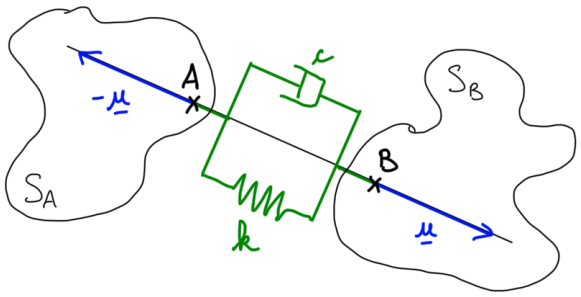

FIGURE 3 – *Liaison de 2 solides par un ressort et un amortisseur*

Le vecteur unitaire qui définit la direction de l'effort est donné par :

$$\underline{u} = \frac{\underline{AB}}{\|\underline{AB}\|} \qquad \text{avec} \qquad \underline{AB}^{\mathscr{R}_0} = \underline{OB}^{\mathscr{R}_0} - \underline{OA}^{\mathscr{R}_0}$$

Le vecteur $\underline{AB}$ peut être calculé très simplement à l'aide de deux appels à la fonction `position`. Le ressort a une longueur initiale l_0. L'élongation δl et la vitesse d'élongation $\dot{\delta l}$ valent :

$$\begin{cases} \delta l = l - l_0 = \|\underline{AB}\| - l_0 \\ \dot{\delta l} = \dot{l} = \dfrac{d}{dt}\|\underline{AB}\| = \dfrac{d}{dt}\sqrt{\underline{AB}^t\ \underline{AB}} = \dfrac{\underline{AB}^t}{\|\underline{AB}\|}\ \dfrac{d}{dt}\underline{AB} = \underline{u}^t\left(\dot{\underline{OB}}^{\mathscr{R}_0} - \dot{\underline{OA}}^{\mathscr{R}_0}\right) \end{cases}$$

La vitesse d'élongation nécessite deux appels à la fonction `vitesse`. La force agissant au point A du solide S_A vaut $\underline{F}_{B\to A}$ (lire "F de B sur A") :

$$\underline{F}_{B\to A} = -\underline{F}_{A\to B} = \left(k\ \delta l + c\ \dot{\delta l}\right)\underline{u}$$

Notez que lorsque $\|\underline{AB}\| = 0$, la direction $\underline{u}$ n'est pas définie. Danc ce cas précis, $\underline{u}$ peut être choisi comme un vecteur unitaire quelconque. En le prenant orthogonal à $\frac{d}{dt}\underline{AB}$, on annule l'effort de l'amortisseur.

Code Python 8.
A la suite du fichier Python **utils2D.py**, programmez les méthodes `deflection` qui renvoie $\underline{AB}$ et $\underline{\dot{AB}}$, `load_direction` qui renvoie $\underline{u}$ et `spring_damper_force` qui renvoie $\underline{F}_{B\to A}$

```
    def deflection(q_a, dq_a, GP_a,
                   q_b, dq_b, GP_b):
        # Relative displacement
        AB = ?___?
        # Relative velocity
        d_AB = ?___?
        return AB, d_AB

    def load_direction(AB, d_AB):
        # Direction of the force
        Long_AB = np.linalg.norm(AB)
        u = ?___?
        return u

    def spring_damper_force(q_a, dq_a, GP_a,
                            q_b, dq_b, GP_b,
                            k, c, L0):
        # Deflection of the spring and dahspot
        AB, d_AB = ?___?
        # Direction of the force
        u = ?___?
        # deflection
        Long_AB = ?___?
        delta   = ?___?
        d_delta = ?___?
        # Force of the spring and the dashpot on point A
        F_BonA = ( k * delta + c * d_delta ) * u
        return F_BonA
```

Considérons le cas où un solide est lié par un ressort/amortisseur à un point B piloté en position et en vitesse dans le référentiel Galiléen. Alors il nous faudra utiliser la fonction `spring_damper_force` en imposant arbitrairement l'évolution du point B : `q_b = [`x_B`, `z_B`, 0]`, `dq_b = [`$\dot{x}_B$`, `$\dot{z}_B$`, 0]` et `GP_b = [0, 0]`.

6 Puissance virtuelle des efforts extérieurs $\mathscr{P}^*$

La puissance virtuelle des efforts extérieurs $\mathscr{P}^*$ est la somme des puissances virtuelles de chaque effort et chaque moment. Par simplicité, nous ne considérerons qu'un unique torseur d'effort appliqué au point P et regroupant l'effort $\underline{F}_{ext} = F_x\underline{X}_0 + F_z\underline{Z}_0$ et le moment $\underline{M}_{ext} = M_y\underline{Y}_0$:

$$\begin{aligned}
\mathscr{P}^* &= \left\{\mathscr{V}_S^{\mathscr{R}_0}\right\}_P^* \otimes \{\mathscr{F}_{ext\to P\in S}\}_P &&= \left\{\underline{\Omega}^*_{S/\mathscr{R}_0} \mid \underline{V}^*_{P\in S/\mathscr{R}_0}\right\}_P^* \otimes \{\underline{F}_{ext} \mid \underline{M}_{ext}\}_P \\
\mathscr{P}^* &= \left(\underline{V}^*_{P\in S/\mathscr{R}_0}\right)^t \underline{F}_{ext} + \left(\underline{\Omega}^*_{S/\mathscr{R}_0}\right)^t \underline{M}_{ext} &&= \left(\underline{\dot{q}}_S^*\right)^t \underline{\underline{T}}_v^t(S,P)\ \underline{F}_{ext} + \left(\dot{\theta}^*\underline{Y}_0\right)^t M_y\underline{Y}_0 \\
\mathscr{P}^* &= P_x\dot{x}^* + P_z\dot{z}^* + P_\theta\dot{\theta}^*
\end{aligned}$$

Ainsi, la puissance virtuelle des efforts extérieurs vaut :

$$\mathscr{P}^* = \left(\underline{\dot{q}}_S^*\right)^t \underline{B} = \langle \dot{x}^*,\ \dot{z}^*,\ \dot{\theta}^*\rangle \left\{\begin{array}{c} P_x \\ P_z \\ P_\theta \end{array}\right\} \qquad \text{avec} \qquad \underline{B} = \underline{\underline{T}}_v^t(S,P)\ \underline{F}_{ext} + \left\{\begin{array}{c} 0 \\ 0 \\ M_y \end{array}\right\} \tag{15}$$

$\underline{B}$ est un vecteur d'effort généralisé.

Code Python 9.
A la suite du fichier Python **utils2D.py**, programmez la méthode `virtual_power` qui renvoie le vecteur d'effort généralisé $\underline{B}$ en fonction de l'effort et du moment subis par un solide.

```
def virtual_power(q, GP_Rs, Fext=np.zeros(2), Mext=0):
    """
    Parameters
    ----------
    q     : numpy array (3, )
    GP_Rs : numpy array (2, )
    Fext  : numpy array (2, ), optional
    The default is np.zeros(2).
    Mext : scalar, optional
    The default is 0.

    Returns
    -------
    B : numpy array (3, )
    """
    ?___?
```

Dans le cas de la pesanteur, la force vaut $F_z = -mg$ et $F_x = 0$. Cette force s'applique au centre de gravité, donc $\underline{GP}^{\mathscr{R}_S} = \underline{GG}^{\mathscr{R}_S} = \underline{0}$. Ainsi, on a $\underline{\underline{T}}_v(S,P) = [\ \underline{\underline{I}},\ \underline{0}\ ]$. De plus, le moment est nul, donc $M_y = 0$. Au final :

$$\underline{B}_{pesanteur} = \left\{ \begin{array}{c} 0 \\ -mg \\ 0 \end{array} \right\} \tag{16}$$

Ainsi, tout solide dispose d'une matrice de masse $\underline{\underline{M}}_S$ et d'un vecteur d'effort généralisé $\underline{B}_S = \underline{B}_{pesanteur}$ pour tenir compte de la pesanteur.

Code Python 10.
Dans la classe `Solid`, ajoutez la méthode `weight` qui renvoie l'effort généralisé $\underline{B}_{pesanteur}$ produit par la pesanteur sur le solide.

```
    def weight(self, g):
        """
        Parameters
        ----------
        g : scalar, 9.81 m/s^2

        Returns
        -------
        B : numpy array (3, )
        """
        ?___?
```

7 Formulation matricielle SANS contraintes

7.1 Assemblage des solides

La formulation matricielle s'obtient par l'égalité $\mathscr{A}^* = \mathscr{P}^*$. Dans le cas d'un unique solide, cela conduit à $(A_x - P_x)\dot{x}^* + (A_z - P_z)\dot{z}^* + (A_\theta - P_\theta)\dot{\theta}^* = 0$. Dans le cas d'un système multicorps, il faut assembler les contributions des N_S différents solides :

$$\underbrace{\left[\begin{array}{ccc|c|c} m_1 & & & & \\ & m_1 & & 0 & \\ & & I_{y1} & & \\ \hline & & & m_2 & \\ 0 & & & & 0 \\ & & & & I_{y2} \\ \hline 0 & & & 0 & \ddots \\ \hline & & & & m_{N_S} \\ 0 & & & 0 & 0 \\ & & & & & I_{yN_S} \end{array}\right]}_{=\underline{\underline{A}}} \underbrace{\left\{\begin{array}{c} \ddot{x}_1 \\ \ddot{z}_1 \\ \ddot{\theta}_1 \\ \hline \ddot{x}_2 \\ \ddot{z}_2 \\ \ddot{\theta}_2 \\ \hline \vdots \\ \hline \ddot{x}_{N_S} \\ \ddot{z}_{N_S} \\ \ddot{\theta}_{N_S} \end{array}\right\}}_{=\underline{\ddot{q}}} = \underbrace{\left\{\begin{array}{c} \sum_k \underline{B}_{k\to S_1} \\ \hline \sum_k \underline{B}_{k\to S_2} \\ \hline \vdots \\ \hline \sum_k \underline{B}_{k\to S_{N_S}} \end{array}\right\}}_{=\underline{B}}$$

On peut l'écrire de manière plus synthétique :

$$\underbrace{\left[\begin{array}{c|c|c|c} \underline{\underline{M}}_1 & & & \\ \hline & \underline{\underline{M}}_2 & & \\ \hline & & \ddots & \\ \hline & & & \underline{\underline{M}}_{N_S} \end{array}\right]}_{=\underline{\underline{A}}} \underbrace{\left\{\begin{array}{c} \underline{\ddot{q}}_1 \\ \hline \underline{\ddot{q}}_2 \\ \hline \\ \hline \underline{\ddot{q}}_{N_S} \end{array}\right\}}_{=\underline{\ddot{q}}} = \underbrace{\left\{\begin{array}{c} \underline{B}_1 \\ \hline \underline{B}_2 \\ \hline \\ \hline \underline{B}_{N_S} \end{array}\right\}}_{=\underline{B}} \tag{17}$$

Dans le cas trivial d'un unique solide en chute libre, on trouve :

$$\left[\begin{array}{ccc} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & I_y \end{array}\right] \left\{\begin{array}{c} \ddot{x} \\ \ddot{z} \\ \ddot{\theta} \end{array}\right\} = \left\{\begin{array}{c} 0 \\ -mg \\ 0 \end{array}\right\}$$

7.2 Intégration temporelle avec Python

Dans Python, pour intégrer des équations différentielles ordinaires (**ODE** = ordinary differential equations) en connaissant leur conditions initiales, on peut utiliser la bibliothèque `scipy.integrate`, voir la documentation [1]. On parlera alors de problèmes aux valeurs initiales (**IVP** = initial value problem). La fonction `scipy.integrate.solve_ivp` [2] regroupe les méthodes d'intégration suivantes :

- 'RK45' : (par défaut) Runge-Kutta explicite d'ordre 5(4)
- 'RK23' : Runge-Kutta explicite d'ordre 3(2)
- 'DOP853' : Runge-Kutta explicite d'ordre 8
- 'Radau' : Runge-Kutta implicite de la famille Radau IIA d'ordre 5
- 'BDF' : méthode implicite multi-pas d'ordre variable (1 à 5)
- 'LSODA' : Méthode Adams/BDF

La documentation précise :

Explicit Runge-Kutta methods ('RK23', 'RK45', 'DOP853') should be used for non-stiff problems and implicit methods ('Radau', 'BDF') for stiff problems. Among Runge-Kutta methods, 'DOP853' is recommended for solving with high precision (low values of rtol and atol).

If not sure, first try to run 'RK45'. If it makes unusually many iterations, diverges, or fails, your problem is likely to be stiff and you should use 'Radau' or 'BDF'. 'LSODA' can also be a good universal choice, but it might be somewhat less convenient to work with as it wraps old Fortran code.

Notez également que les méthodes implicites ('Radau', 'BDF' et 'LSODA') requièrent de fournir la matrice jacobienne du second membre. Cette jacobienne peut être pénible à calculer, malheureusement, les problèmes multicorps sont souvent des problèmes raides (stiff) qui nécessitent des méthodes implicites et donc des jacobiennes !

7.3 Modèle d'état

Les algorithmes d'intégration temporelle cités précédemment sont du premier ordre : ils traitent uniquement des fonctions du type $\dot{\underline{y}} = \underline{f}(t, \underline{y})$. Or nous aimerions traiter le cas $\ddot{\underline{q}} = \underline{f}(t, \underline{q}, \dot{\underline{q}})$, alors on passe par un modèle d'état du premier ordre. Notons N_{ddl} le nombre de degrés de libertés, ainsi $\underline{\underline{A}}$ est de taille $N_{\text{ddl}} \times N_{\text{ddl}}$:

$$\begin{bmatrix} \underline{\underline{I}}_{N_{\text{ddl}}} & \underline{\underline{0}}_{N_{\text{ddl}}} \\ \underline{\underline{0}}_{N_{\text{ddl}}} & \underline{\underline{A}} \end{bmatrix} \begin{Bmatrix} \dot{\underline{q}} \\ \ddot{\underline{q}} \end{Bmatrix} = \begin{Bmatrix} \dot{\underline{q}} \\ \underline{B} \end{Bmatrix} \tag{18}$$

Utilisons la notation :

$$\underline{y} = \begin{Bmatrix} \underline{q} \\ \dot{\underline{q}} \end{Bmatrix} \quad \Longrightarrow \quad \dot{\underline{y}} = \begin{Bmatrix} \dot{\underline{q}} \\ \ddot{\underline{q}} \end{Bmatrix} = \begin{Bmatrix} \dot{\underline{q}} \\ \underline{\underline{A}}^{-1}\underline{B} \end{Bmatrix}$$

1. https://docs.scipy.org/doc/scipy/reference/integrate.html#module-scipy.integrate
2. https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html#scipy.integrate.solve_ivp

7.4 Exemples codés à la main

7.4.1 Un solide en chute libre

Soit un solide de masse $m = 1$ kg, d'inertie $I_y = 1\ \mathrm{kg\,m^2}$ ayant pour conditions initiales :

$$\underline{q}(0) = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \quad \text{et} \quad \dot{\underline{q}}(0) = \begin{Bmatrix} 1 \\ 1 \\ 0 \end{Bmatrix}$$

La solution analytique servira de référence pour valider le modèle numérique Matlab. Elle vaut :

$$\begin{cases} m\ddot{x} = 0 \\ m\ddot{z} = -mg \\ I_y\ddot{\theta} = 0 \end{cases} \implies \begin{cases} \dot{x} = 1 \\ \dot{z} = -gt + 1 \\ \dot{\theta} = 0 \end{cases} \implies \begin{cases} x = t \\ z = -\frac{1}{2}gt^2 + t \\ \theta = 0 \end{cases}$$

Exercice 2.
Intégrez avec la fonction `scipy.integrate.solve_ivp` la dynamique de ce solide en chute libre pour $t \in [0\ ,\ 0.25]$ s. Puis superposez sur une même figure la solution numérique avec la solution analytique.

Correction de l'exercice 2.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as sci

# Parametres
Ne = 50
g = 9.81

# Solution theorique
t = np.linspace(0, 0.25, Ne)
x = t
z = - 1/2 * g * t**2 + t
theta = np.zeros(Ne)

# matrice de masse et vecteur de pesanteur
m = 1
Iy = 1
M = np.diag([m, m, Iy])
B = np.array([0, -m*g, 0])
args = (M, B)

# Derivee de l'etat
def fun(t, y, M, B):
    d2q = np.linalg.solve(M, B)
    dy = np.concatenate((y[3:6], d2q))
return dy
```

```
# Integration temporelle avec Runge Kutta
t_span = [0, 0.25]
y0 = np.array([0, 0, 0, 1, 1, 0])
sol = sci.solve_ivp(fun, t_span, y0, t_eval=t, args=args)
t_rk = sol.t
x_rk = sol.y[0]
z_rk = sol.y[1]
theta_rk = sol.y[2]

# Resultats
ax_x = plt.subplot(3, 1, 1)
ax_x.grid()
ax_x.plot(t, x, label='theorie')
ax_x.plot(t_rk, x_rk, 'o', label='solve ivp')
ax_x.legend()

ax_z = plt.subplot(3, 1, 2)
ax_z.grid()
ax_z.plot(t, z, label='theorie')
ax_z.plot(t_rk, z_rk, 'o', label='solve ivp')
ax_z.legend()

ax_theta = plt.subplot(3, 1, 3)
?___?
```

7.4.2 Un pendule sur ressort

Soit un solide de masse $m = 1$ kg, d'inertie $I_y = 10^{-2}$ kg m^2. Ce solide est accroché par un ressort situé entre le point P du solide et le point A du repère Galiléen. Ce ressort a une longueur $l_0 = 0.5$, une raideur $k = 10^3$ N/m et un amortissement $c = 10^2$ N s/m.

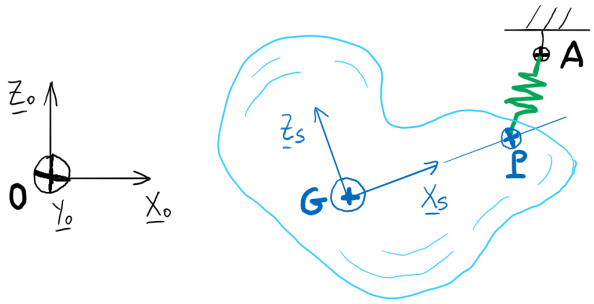

FIGURE 4 – *Pendule accroché par un ressort*

Les positions des points G, P et A dans le repère Galiléen valent :

$$\underline{OG}^{\mathscr{R}_0} = \left\{ \begin{array}{c} 1 \\ 3 \end{array} \right\} \quad , \quad \underline{OP}^{\mathscr{R}_0} = \left\{ \begin{array}{c} 2 \\ 4 \end{array} \right\} \quad \text{et} \quad \underline{OA}^{\mathscr{R}_0} = \left\{ \begin{array}{c} 3 \\ 2 \end{array} \right\}$$

Pour calculer les conditions initiales du système, il nous manque l'angle θ du solide. On peut le choisir arbitrairement. Ici, nous souhaitons que le vecteur $\underline{GP}$ soit colinéaire au vecteur $\underline{X}_S$ du repère du solide. Cela nous contraint sur le choix de l'angle θ.

$$\underline{GP}^{\mathscr{R}_0} = \underline{OP}^{\mathscr{R}_0} - \underline{OG}^{\mathscr{R}_0} \quad \Longrightarrow \quad \underline{X}_S = \frac{\underline{GP}^{\mathscr{R}_0}}{\|\underline{GP}^{\mathscr{R}_0}\|} = \left\{ \begin{array}{c} \alpha \\ \beta \end{array} \right\}$$

$$\Longrightarrow \quad \tan(\theta) = \frac{-\beta}{\alpha} \quad \text{et} \quad \underline{Z}_S = \left\{ \begin{array}{c} -\beta \\ \alpha \end{array} \right\}$$

Nous choisissons de lacher le pendule avec une vitesse nulle. Ainsi, les conditions initiales en position et vitesse valent :

$$\underline{q}(0) = \left\{ \begin{array}{c} \underline{OG}^{\mathscr{R}_0} \\ \theta \end{array} \right\} \quad \text{et} \quad \underline{\dot{q}}(0) = \left\{ \begin{array}{c} 0 \\ 0 \\ 0 \end{array} \right\}$$

Il reste à déterminer la position du point P dans le repère du solide. Pour cela nous pouvons inverser l'équation (2) :

$$\underline{GP}^{\mathscr{R}_S} = \left(\underline{\underline{R}}_{0 \leftarrow S}\right)^{-1} \left(\underline{OP}^{\mathscr{R}_0} - \underline{OG}^{\mathscr{R}_0}\right) = \underline{\underline{R}}^t_{0 \leftarrow S} \, \underline{GP}^{\mathscr{R}_0}$$

Exercice 3.
Intégrez avec la fonction `scipy.integrate.solve_ivp` la dynamique de ce solide pour $t \in [0\ ,\ 5]$ s. Tracez l'état (x, z et θ) du solide en fonction du temps.

Correction de l'exercice 3.

```
import numpy as np
import scipy.integrate as sci
import matplotlib.pyplot as plt
import matplotlib.animation as ani
from utils2D import rotation, position, \
    spring_damper_force, virtual_power

# Position des points dans le repere galileen
OG_0 = np.array([1, 3])
OP_0 = np.array([2, 4])
OA_0 = np.array([3, 2])

# Orientation initial du solide
GP_0 = OP_0 - OG_0
X_S = GP_0 / np.linalg.norm(GP_0)
theta_0 = - np.arctan2(X_S[1], X_S[0])

# Etat initial du solide
q_0 = np.concatenate((OG_0, [theta_0]))

# Position du point P dans le repere du solide
R_0S = rotation(q_0)
GP_S = R_0S.T @ GP_0

# matrice de masse et vecteur de pesanteur
m = 1
Iy = 0.01
g = 9.81
M = np.diag([m, m, Iy])
B_poids = np.array([0, -m*g, 0])

# Loi de comportement du ressort/amortisseur
k = 1e3
c = 1e2
L0 = 0.5

# spring_damper_force() renvoie F_BonA, ici le point A demande
# par spring_damper_force() correspond au point P du solide S
# et le point B sera attache au repere Galileen
GP_a = GP_S
q_b = np.concatenate((OA_0, [0]))
dq_b = np.zeros(3)
GP_b = np.zeros(2)

# Integration temporelle avec Runge Kutta
```

```
    # Liste des arguments supplementaires a passer a la fonction fun
    args = (M, B_poids, GP_a, q_b, dq_b, GP_b, k, c, L0)
    def fun(t, y, M, B_poids, GP_a, q_b, dq_b, GP_b, k, c, L0):
        q_a = y[0:3]
        dq_a = y[3:]
        F_BonA = spring_damper_force(q_a, dq_a, GP_a,
                                     q_b, dq_b, GP_b,
                                     k, c, L0)
        B_ressort = virtual_power(q_a, GP_a, Fext=F_BonA)
        B = B_poids + B_ressort
        d2q = np.linalg.solve(M, B)
        dy = np.concatenate((y[3:6], d2q))
        return dy

    Ne = 1000
    tf = 5
    t = np.linspace(0, tf, Ne)
    y0 = np.concatenate( (q_0, np.zeros(3)) )
    sol = sci.solve_ivp(fun, [0, tf], y0, t_eval=t, args=args)

    # Resultats
    t = sol.t
    x = sol.y[0]
    z = sol.y[1]
    theta = sol.y[2]

    # Evolution temporelle
    ax_x = plt.subplot(3, 1, 1)
    ax_x.grid()
    ax_x.plot(t, x, label='solve ivp')
    ax_x.set_ylabel('Position x [m]')
    ax_x.legend()

    ax_z = plt.subplot(3, 1, 2)
    ax_z.grid()
    ax_z.plot(t, z, label='solve ivp')
    ax_z.set_ylabel('Altitude z [m]')
    ax_z.legend()

    ax_theta = plt.subplot(3, 1, 3)
    ax_theta.grid()
    ax_theta.plot(t, theta, label='solve ivp')
    ax_theta.set_xlabel('Temps [s]')
    ax_theta.set_ylabel('Angle [rad]')
    ax_theta.legend()

    # Animation
    fig, ax_anim = plt.subplots()
    ax_anim.axis('equal')
    ax_anim.set_xlim(left=OA_0[0]-3, right=OA_0[0]+3)
    ax_anim.set_ylim(bottom=OA_0[1]-3, top=OA_0[1]+3)
    ax_anim.grid()
    titre = ax_anim.set_title("t = {:.3f}s".format(0))
```

```
curv1, = ax_anim.plot(np.zeros(2), np.zeros(2),
                      marker='^', label="solid")
curv2, = ax_anim.plot(np.zeros(2), np.zeros(2),
                      marker='+', label="ressort")
ax_anim.legend()

# Creating an animation
def animate_fct(ind):
    titre.set_text("t = {:.3f}s".format(t[ind]))
    OP_t = position(sol.y[0:3, ind], GP_S)
    # Animation du solide : segment GP_0
    x1 = np.array( [ x[ind], OP_t[0] ] )
    z1 = np.array( [ z[ind], OP_t[1] ] )
    curv1.set_data(x1, z1)
    # Animation du ressort : segment AP_0
    x2 = np.array( [ OA_0[0], OP_t[0] ] )
    z2 = np.array( [ OA_0[1], OP_t[1] ] )
    curv2.set_data(x2, z2)
    return curv1, curv2

anim = ani.FuncAnimation(fig=fig, func=animate_fct,
                         frames=len(t)-1, interval=10)
```

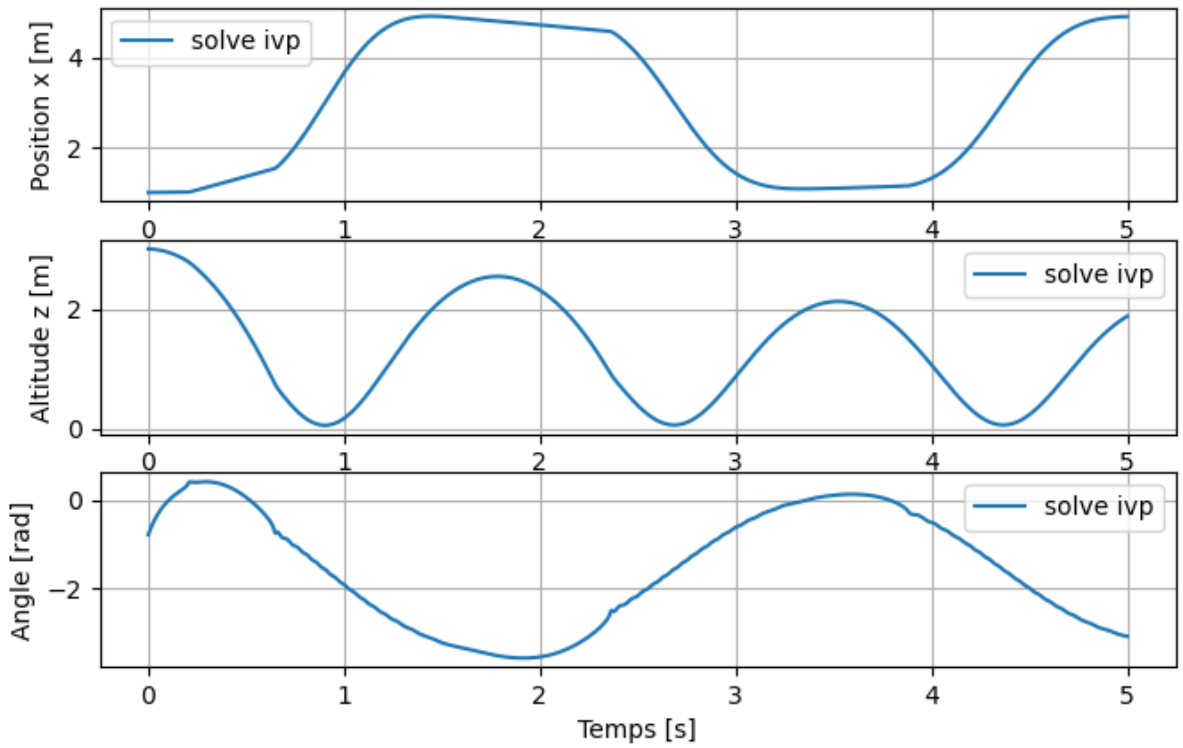

FIGURE 5 – *Evolution temporelle du pendule accroché à un ressort*

7.5 Impémentation dans Python

7.5.1 Cahier des charges

Les deux exemples précédents nous ont permis d'avoir un aperçu des tâches que l'on aimerait que le programme Python effectue automatiquement :

- Calcul de l'état initial des solides : $\underline{q}_S(t_0) = \langle x_0, z_0, \theta_0 \rangle$ et $\underline{\dot{q}}_S(t_0) = \langle \dot{x}_0, \dot{z}_0, \dot{\theta}_0 \rangle$
- Calcul de la position des points de liaison dans le repère du solide : $\underline{GP}_S^{\mathscr{R}}$
- Calcul de la matrice de masse de chaque solide : $\underline{\underline{M}}_S$
- Calcul de l'effort généralisé de pesanteur de chaque solide : $\underline{B}_{pesanteur}$
- Définition de l'ordre des degrés de libertés des solides dans le vecteur d'état : $\underline{y} = \langle \underline{q}, \underline{\dot{q}} \rangle$ et $\underline{q} = \langle \underline{q}_1, ..., \underline{q}_N \rangle$. Plus généralement, affectation d'indices à chaque solide et à chaque interaction.
- Concatenation de la matrice de masse de tous les solides $\underline{\underline{M}}$
- Calcul des efforts généralisés de chaque interaction et concatenation du vecteur $\underline{B}$

Voyons maintenant comment architecturer le programme Python afin d'automatiser les tâches listées précédemment. Il existe des paramètres globaux tels que la constante de gravité $g = 9.81\text{m/s}^2$, nous les stockerons comme des propriétés de la classe `Param`. L'intégration temporelle nécessite également un certains nombre de matrices et de vecteurs temporaires que nous stockerons comme des propriétés de la classe `Matrices`.

Code Python 11.
A la suite du fichier Python **Multicorps2D.py**
Programmez les classes `Param` et `Matrices`

```
class Param:
    def __init__(self):
        self.g = 9.81

class Matrices:
    def __init__(self):
        pass
```

L'équation (17) met en jeu les caractéristiques des solides pour calculer la matrice $\underline{\underline{A}}$ et les interactions entre les solides pour calculer le vecteur $\underline{B}$. Les interactions entre solides peuvent être de nature relativement différente :

- des efforts extérieurs appliqués à un seul solide à la fois
- des lois de comportement entre plusieurs solides
- des contraintes cinématiques entre solides (pivot, glissière, ...)
- ...

Malheureusement, il n'est pas possible de définir une seule et unique classe qui convienne à tous les types d'interactions. En effet, certaines interactions auront besoin des données provenant d'un seul solide, quand d'autres lieront plusieurs solides à la fois. Certaines auront besoin d'un paramètre de raideur, d'autres non, etc. En revanche toutes ces différentes interactions auront des caractéristiques communes telles que les propriétés suivantes :

- `label` : le nom de l'interaction
- `n_c` : le nombre d'équations de contraintes, voir la section 8.1. Dans le cas d'un interaction du type ressort/amortisseur telle que décrite en section 5.2, $n_c = 0$
- `ind_c` : indice des equations de contraintes dans les matrices, voir la section 8.1.
- `tag_S` : la liste des tags des `Solid` rentrant en jeu dans l'interaction
- `tag_F` : la liste des tags des `Frame` rentrant en jeu dans l'interaction.

Toutes les classes d'interactions devront comporter une méthode `assemble()` qui sera appelée lors de l'intégration temporelle de l'équation (17). Cette méthode calculera les vecteurs d'efforts généralisés $\underline{B}_{k\to S_i}$ qui la concernent. L'interaction devra également placer ces vecteurs d'efforts généralisés au bon endroit dans le vecteur global $\underline{B}$.

7.5.2 La première interaction : `SpringDashpot_2s`

Pour l'instant, la seule interaction que l'on soit en mesure de programmer est celle décrite dans la section 5.2. Elle sera implémentée dans la classe `SpringDashpot_2s`. Elle met en jeu 2 solides différents. Sa méthode `__init__()` permet de stocker tous les paramètres nécessaires au calcul de son effort généralisé. Notamment, il est nécessaire de connaitre les indices des degrés de libertés du solide A (`ind_q_a` et `ind_dq_a`) et du solide B (`ind_q_b` et `ind_dq_b`) mais aussi les points extrémités du ressort accrochés au solide A (`GP_a`) et au solide B (`GP_b`). La méthode `_build()` de la classe `SpringDashpot_2s` permettra justement de récupérer ces informations. Nous y reviendrons plus loin.

Code Python 12.
A la suite du fichier Python **Multicorps2D.py**
Programmez la classe `SpringDashpot_2s`

```
class SpringDashpot_2s:
    def __init__(self, label="SpringDashpot2S",
                 tag_S=["tag_Sa", "tag_Sb"],
                 tag_F=["tag_Fa", "tag_Fb"],
                 k=1e3, c=1, L0=0.5):
        # Common properties
        self.label  = label  # Label of the interaction
        self.n_c    = 0      # Number of contraints equations
        self.ind_c  = 0      # Indices of contraints equations
        self.tag_S  = tag_S  # Tags of the solids in interaction
        self.tag_F  = tag_F  # Tags of the frames in interaction
        # Specific properties
        # Constitutive law
        self.k = k
        self.c = c
        self.L0 = L0
        # Parameters of Solid A
```

```
            self.ind_q_a = []
            self.ind_dq_a = []
            self.GP_a = []
            # Parameters of Solid B
            self.ind_q_b = []
            self.ind_dq_b = []
            self.GP_b = []

        def _build(self, solids, param):
            pass

        def assemble(self, t, y, matrices):
            pass
```

7.5.3 La classe `Model`

Notre modèle multicorps est donc composé de solides et d'interactions entre ces mêmes solides. Chaque solide comporte lui-même un certain nombre de repères. Les interactions lient un ou plusieurs solides en des points particuliers définis par ces repères. Ainsi, il est nécessaire de créer une classe `Model` qui comportera un dictionnaire de `Solid` et une dictionnaire d'`Interaction`. Elle aura aussi pour propriétés une classe `Param` et une classe `Matrices`.

La classe `Model` comportera dans un premier temps les méthodes suivantes :

- `add_solid()` qui permettra d'ajouter un `Solid` au dictionnaire des solides en lui donnant une clé grâce à la variable `tag_S`.
- `add_frame()` qui appellera la méthode `add_frame()` de la classe `Solid`. Cela permet de gérer les solides depuis la classe `Model` sans avoir à fouiller dans le dictionnaire de solides.
- `add_interaction()` qui permettra d'ajouter une `Interaction` au dictionnaire des interactions en lui donnant une clé grâce à la variable `tag_I`.
- `build()` qui affectera des indices à chaque solide et à chaque interaction.
- `transient()` qui permettra de réaliser l'intégration temporelle du modèle multicorps grâce à `scipy.integrate.solve_ivp()`. Elle fera appelle à la méthode `_odefun()`.
- `_odefun()` permettra de calculer la dérivée temporelle de l'état.

Code Python 13.
A la suite du fichier Python **Multicorps2D.py**
Programmez la classe `Model`

```
class Model:
    def __init__(self):
        # Default parameters of the model
        self.param = Param()

        # Empty dictionaries of solids
        self.solids = {}

        # Empty dictionaries of interactions
        self.interactions = {}

        # Number of
```

```
        self.n_s = 0  # solids
        self.n_q = 0  # dof (degree of freedom)
        self.n_i = 0  # interactions
        self.n_c = 0  # constraints

        # Some matrices for later computation
        self._matrices = Matrices()

    def add_solid(self, tag_S="solid", label="solid",
                  m=1.0, iy=0.01,
                  G_t0=np.zeros(2), Xs=np.array([1., 0.]),
                  dG_t0=np.zeros(2), dtheta_t0=0,
                  color=[0, 0, 1], marker='+'):
        self.solids[tag_S] = Solid(?___?)

    def add_frame(self, tag_S="solid", tag_F="frame",
                  label="frame", P_t0=np.zeros(2),
                  Xs=np.array([1., 0.])):
        self.solids[tag_S].add_frame(?___?)

    def add_interaction(self, kind,
                        tag_I="interaction", **kwargs):
        if kind == "SpringDashpot_2s":
            inter = SpringDashpot_2s(**kwargs)
        # Creation of the interaction
        self.interactions[tag_I] = inter

    def build(self):
        pass

    def _odefun(self, t, y):
        ind_q = ?___?
        ind_dq = ?___?

        # Matrices assembly
        matrice = ?___?
        vecteur = ?___?

        # d2q = inv(M) * F
        d2q = np.linalg.solve(matrice, vecteur)
        dy = np.concatenate((y[ind_dq], d2q))
        return dy

    def transient(self, t0, tf, dt,
                  method='RK45', max_step=np.inf,
                  rtol=1e-3, atol=1e-6):
        # Updating of the model before time integration
        self.build()

        # Parameters for time integration
        fun = self._odefun
        t_span = [t0, tf]
        y0 = self._matrices.y0
```

```
        t_eval = np.arange(t0, tf, dt)

        # Time integration itself
        sol = sci.solve_ivp(fun, t_span, y0,
                            method=method,
                            t_eval=t_eval, #vectorized=False,
                            max_step=max_step,
                            rtol=rtol, atol=atol)

        return sol
```

Le vecteur $\underline{B}$ des efforts généralisés change à chaque instant et doit être recalculé par la méthode `Model._odefun()`. Or la taille et la composition de ce vecteur dépend du nombre de solides et des interactions entre eux. Ainsi, le calcul des efforts généralisés et leur assemblage est délégué aux classes `Interaction` telle que `SpringDashpot_2s`. En pratique la méthode `Model._odefun()` appelle les méthodes `Interaction.assemble()` de chaque interaction dans une boucle `for`.

Code Python 14.
Implémentez la méthode `assemble` de la classe `SpringDashpot_2s`

```
    def assemble(self, t, y, matrices):
        # Vector AB = 0
        # with A the point defined by tag_F in Solid tag_S
        # with B the fixed point in the galilean frame
        # Parameters of Solid A
        q_a = y[self.ind_q_a]
        dq_a = y[self.ind_dq_a]
        GP_a = self.GP_a
        # Parameters of Solid B
        q_b = y[self.ind_q_b]
        dq_b = y[self.ind_dq_b]
        GP_b = self.GP_b

        # Force of the spring and the dashpot on point A
        F_BonA = spring_damper_force(q_a, dq_a, GP_a,
                                     q_b, dq_b, GP_b,
                                     self.k, self.c, self.L0)

        # Matrix assembly
        ind_a = np.ix_(self.ind_q_a)
        matrices.B[ind_a] += virtual_power(Fext=F_BonA, Next=0,
                                           q=q_a, GP_Rs=GP_a)
        ind_b = np.ix_(self.ind_q_b)
        matrices.B[ind_b] += virtual_power(Fext=-F_BonA, Next=0,
                                           q=q_b, GP_Rs=GP_b)
        return matrices
```

7.5.4 Initialisation du modèle avec les méthodes `_build()`

Pour que la méthode `Interaction.assemble()` puisse placer ses efforts généralisés au bon endroit dans le vecteur global $\underline{B}$, il faut qu'elle connaisse les indices des degrés de libertés du solide A (`ind_q_a` et `ind_dq_a`) et du solide B (`ind_q_b` et `ind_dq_b`) mais aussi les points extrémités du ressort accrochés au solide A (`GP_a`) et au solide B (`GP_b`).

Ces indices dépendent du nombre de solide et de l'ordre de leur création. Avant tout calcul, il sera donc nécessaire d'inspecter ce que contient réellement le modèle pour déterminer ces indices. Cette opération sera réalisée automatiquement par la méthode `build()` de la classe `Model`. Cette méthode réalisera les opérations suivantes :

1. `Model._build_solids()` : Affectation d'un vecteur d'indice de position `ind_q` et d'un vecteur d'indice de vitesse `ind_dq` à chaque solide du dictionnaire et initialisation des matrices et des vecteurs du problème.
2. `Model._build_interactions()` : Appel de `Interaction._build()` de chaque classe d'interaction pour qu'elle prenne note des indices des solides qui la concernent.
3. `Model._build_initial_conditions()` : Concaténation des états initiaux des solides pour créer le vecteur d'état global initial $\underline{q}_0$.

Il faut donc modifier la classe `Model` comme suit :

Code Python 15.
Ajoutez les méthodes suivantes à la classe `Model`

```
    def build(self):
        # Updating of :
        # + the indices of degrees of freedom of solids,
        # + matrices A, B, C and D
        # + interactions
        # + initial conditions
        self._build_solids()
        self._build_interactions()
        self._build_initial_conditions()

    def _build_solids(self):
        # Number of solids in the model
        self.n_s = len(self.solids)
        self.n_q = 0  # Length of vector q

        # Initialisation of matrices A and B
        A = []
        B = []
        ind_q = []
        for s in self.solids.values():
            # List of mass matrices to be concatenated
            M = s.mass()
            A.append(M)
            # List gravity vectors to be concatenated
            F = s.weight(self.param.g)
            B.append(F)
```

```
            # Indices of the position dofs of the current solid
            s.ind_q = self.n_q + np.arange(0, 3)
            ind_q.append(s.ind_q)
            # Updating of the total number of position dofs
            self.n_q += 3

        # Indices of the velocity dofs
        ind_dq = []
        for s in self.solids.values():
            s.ind_dq = self.n_q + s.ind_q
            ind_dq.append(s.ind_dq)

        # Matrices A, B, C and D
        self._matrices.A        = scl.block_diag(*A)
        self._matrices.B        = np.concatenate(B)
        self._matrices.y0       = np.zeros(2*self.n_q)
        self._matrices.y        = np.zeros(2*self.n_q)
        self._matrices.ind_q    = np.concatenate(ind_q)
        self._matrices.ind_dq   = np.concatenate(ind_dq)

    def _build_interactions(self):
        # Number of interactions in the model
        self.n_i = len(self.interactions)
        self.n_c = 0  # Length of vector lambda
        for i in self.interactions.values():
            i._build(self.solids, self.param)
            # Indices of the position constraints in matrix C
            i.ind_c = self.n_c + np.arange(0, i.n_c)
            # Updating of the total number of constraints
            self.n_c += i.n_c

        # Matrices A, B, C and D
        n_c = self.info.n_c
        n_q = self.info.n_q
        self._matrices.C        = np.zeros((n_c, n_q))
        self._matrices.D        = np.zeros(n_c)
        self._matrices.ZerC     = np.zeros((n_c, n_c))

    def _build_initial_conditions(self):
        y0 = np.zeros( 2 * self.n_q )
        for s in self.solids.values():
            y0[s.ind_q] = s.q_t0
            y0[s.ind_dq] = s.dq_t0
        self._matrices.y0 = y0
```

Nous venons de voir que `Model.build()` appelle `Model._build_interactions()` qui appelle ensuite `SpringDashpot_2s._build()`. Il faut donc implémenter la méthode `_build()` de la classe `SpringDashpot_2s` afin qu'elle récupère et stocke les paramètres suivants : `ind_q_a`, `ind_dq_a`, `GP_a`, `ind_q_b`, `ind_dq_b` et `GP_b`.

Code Python 16.
Implémentez la méthode `_build()` de la classe `SpringDashpot_2s`

```
    def _build(self, solids, param):
        # Solid Sa
        s_a = solids[self.tag_S[0]]
        f_a = s_a.frames[self.tag_F[0]]
        self.ind_q_a = s_a.ind_q
        self.ind_dq_a = s_a.ind_dq
        self.GP_a = f_a.position
        # Solid Sb
        s_b = solids[self.tag_S[1]]
        f_b = s_b.frames[self.tag_F[1]]
        self.ind_q_b = s_b.ind_q
        self.ind_dq_b = s_b.ind_dq
        self.GP_b = f_b.position
```

Exercice 4.
Refaire l'exercice 2 du solide en chute libre en utilisant la nouvelle implémentation du code Python. Superposez sur une même figure la nouvelle solution numérique avec celle obtenue à l'exercice 2.

La classe `SpringDashpot_2s` lient 2 solides entre eux. Or, dans l'exercice 3 du pendule sur ressort, un unique solide est lié au repère galiléen. Nous avons donc besoin d'une interaction de classe `SpringDashpot_1sFix` qui lie un solide au repère galiléen.

Code Python 17.
A la suite du fichier Python **Multicorps2D.py**
Implémentez la classe `SpringDashpot_1sFix`

```
    class SpringDashpot_1sFix:
        def __init__(self, label="SpringDashpot_1sFix",
                     tag_S="tag_S", tag_F="tag_F",
                     P_Gal=np.array([0, 0]),
                     k=1e3, c=1, L0=0.5):
            # Common properties
            self.label  = label  # Label of the interaction
            self.n_c    = 0      # Number of contraints equations
            self.tag_S  = tag_S  # Tags of the solids in interaction
            self.tag_F  = tag_F  # Tags of the frames in interaction
            # Specific properties
            # Constitutive law
            self.k = k
```

```
        self.c = c
        self.L0 = L0
        # End point of the spring in the Galilean frame
        self.P_Gal = P_Gal
        # Parameters of Solid A
        self.ind_q_a = []
        self.ind_dq_a = []
        self.GP_a = []

    def _build(self, solids, param):
        self.ind_q_a = ?___?
        self.ind_dq_a = ?___?
        self.GP_a = ?___?

    def assemble(self, t, y, matrices):
        # Force of the spring and the dashpot on point A
        F_BonA = spring_damper_force(?___?)

        # Matrix assembly
        ind_a = np.ix_(self.ind_q_a)
        matrices.B[ind_a] += virtual_power(Fext=F_BonA, ?___?)
```

Exercice 5.
Refaire l'exercice 3 du pendule sur ressort en utilisant la nouvelle implémentation du code Python. Superposez sur une même figure la nouvelle solution numérique avec celle obtenue à l'exercice 3.

7.6 Quelques interactions supplémentaires

7.6.1 Une interaction pour les efforts et les moments extérieurs

Pour imposer un effort extérieur, il faut connaitre :

- son évolution temporelle : effort constant, défini analytiquement ou interpolé à partir d'un tableau de valeurs
- le repère dans lequel il est défini : le repère galiléen constant ou un repère attaché à un solide qui dépend du temps

Commençons par un fonction `tabular` pour interpoler un tableau à 2 lignes. `tabular` renverra un vecteur à un instant `t` donné

Code Python 18.
Dans **utils2D.py**

```
def tabular(t, t_tab, f_tab):
    f_ext = np.zeros(2)
    f_ext[0] = np.interp(t, t_tab, f_tab[0])
    f_ext[1] = np.interp(t, t_tab, f_tab[1])
    return f_ext
```

Exercice 6.
Implémentez une classe d'interaction qui permette d'imposer un effort sur un point d'un solide. Cette classe aura un argument d'entrée `fun` de type "callable" c'est-à-dire une fonction python. Sa signature sera `fun(t)`. Elle renverra un vecteur `numpy.array` à 2 composantes correspondant à $\underline{F}_{ext}$ dans l'équation (15).

```
class Force:
    def __init__(self, label="Force",
                 tag_S="tag_S", tag_F="tag_F",
                 fun=tabular()):
        ?___?

    def _build(self, solids, param):
        ?___?

    def assemble(self, t, y, matrices):
        ?___?
```

Elle pourra par exemple être utilisée en combinaison avec la fonction `tabular` précédente en passant par une "fonction lambda" de Python de la façon suivante :

```
t_tab = np.linspace(0, 5, num=501)
f_tab = np.array( (np.zeros(501), 1e3*np.sin(t_tab)) )
fonc = lambda t : tabular(t, t_tab, f_tab)
interaction = Force(tag_S="S1", tag_F="P", fun=fonc)
```

Exercice 7.
Implémentez une classe d'interaction qui permet d'imposer un moment sur un solide.

```
class Moment:
    def __init__(self, label="Force", tag_S="tag_S"):
        ?___?

    def _build(self, solids, param):
        ?___?

    def assemble(self, t, y, matrices):
        ?___?
```

7.6.2 Une interaction naïve pour simuler le contact

Ici nous allons chercher à modéliser le contact au sol d'un solide. Pour simplifier, nous allons considérer que :

- le sol est plat et situé à une altitude z_0
- seul un point rentrera en contact avec le sol
- lors de l'interpénétration, l'effort engendré sera de la forme $k(z - z_0) + c\dot{z}$

Exercice 8.
Implémentez une classe d'interaction entre un solide et le sol.

```
class Contact:
    def __init__(self, label="Contact au sol",
                 tag_S="tag_S", tag_F="tag_F",
                 z0=0, k=1e3, c=1e2):
        ?___?

    def _build(self, solids, param):
        ?___?

    def assemble(self, t, y, matrices):
        ?___?
```

8 Formulation matricielle AVEC contraintes

8.1 Typologie des contraintes

Dans cette partie nous allons voir comment ajouter des contraintes cinématiques sur le modèle mécanique. Cela permet de modéliser des liaisons parfaites mais aussi d'imposer un mouvement arbitraire à un point d'un solide.

Toute contrainte sera écrite sous la forme $\underline{g}(\underline{q}, \underline{\dot{q}}, t) = \underline{0}$. Côté vocabulaire, on distinguera les contraintes en fonction de leur dépendance en vitesse (holonome ou non) et de leur dépendance au temps (rhéonome ou scléronome) :

$\underline{g}(\underline{q})$	holonome	et	scléronome :	pivot, rotule, glissière, ...
$\underline{g}(\underline{q}, \underline{\dot{q}})$	non-holonome	et	scléronome :	le roulement sans glissement, ...
$\underline{g}(\underline{q}, t)$	holonome	et	rhéonome :	le pilotage de la position d'un point, ...
$\underline{g}(\underline{q}, \underline{\dot{q}}, t)$	non-holonome	et	rhéonome :	...

Inclure $\underline{g} = \underline{0}$ dans l'équation différentielle revient à transformer le système ODE [3] (Ordinary Differential Equation) en système DAE [4] (Differential Algebraic Equation). Or peu d'algorithmes d'intégration sont capables de traiter les systèmes DAE. En revanche, en dérivant la contrainte par rapport au temps, jusqu'à faire apparaitre l'accélération, on reste sur un système ODE. Pour cela il faut dériver deux fois les contraintes holonomes et une seule fois les contraintes non-holonomes. On peut alors écrire la contrainte sous la forme :

$$\begin{aligned} \text{holonome} \quad & \underline{\ddot{g}} = \underline{\underline{C}}\ \underline{\ddot{q}} - \underline{D} = \underline{0} \\ \text{non-holonome} \quad & \underline{\dot{g}} = \underline{\underline{C}}\ \underline{\ddot{q}} - \underline{D} = \underline{0} \end{aligned} \qquad \Longrightarrow \qquad \boxed{\underline{\underline{C}}\ \underline{\ddot{q}} = \underline{D}} \tag{19}$$

Lors de l'intégration temporelle par `scipy.integrate.solve_ivp`, les approximations numériques combinées à la dérivation des contraintes font apparaitre des constantes d'intégration. Cela créera une dérive entrainant le non-respect de la contrainte initiale $\underline{g} \neq \underline{0}$. Pour éviter cela on utilise la stabilisation de Baumgarte. Cela consiste à introduire la contrainte dans une équation différentielle, de sorte qu'elle soit de nouveau respectée au bout d'un temps t_{regul} :

$$\begin{aligned} \text{holonome} \quad & \underline{\ddot{g}} + 2\xi_{\text{bg}}\omega_{\text{bg}}\underline{\dot{g}} + \omega_{\text{bg}}^2\underline{g} = \underline{0} \quad \text{avec} \quad \xi_{\text{bg}} = 1 \quad \text{et} \quad \omega_{\text{bg}} = \frac{2\pi}{t_{\text{regul}}} = 2\pi f_{\text{regul}} \\ \text{non-holonome} \quad & \underline{\dot{g}} + \frac{1}{\tau_{\text{bg}}}\underline{g} = \underline{0} \quad \text{avec} \quad \tau_{\text{bg}} = t_{\text{regul}} \end{aligned}$$

Ainsi, la méthode de Baumgarte revient à ajouter des termes de stabilisation dans le membre de droite de l'équation de contrainte :

$$\begin{aligned} \text{holonome} \quad & \underline{D}_{\text{bg}} = \underline{D} - 2\xi_{\text{bg}}\omega_{\text{bg}}\underline{\dot{g}} - \omega_{\text{bg}}^2\underline{g} \\ \text{non-holonome} \quad & \underline{D}_{\text{bg}} = \underline{D} - \frac{1}{\tau_{\text{bg}}}\underline{g} \end{aligned}$$

3. = équation différentielle ordinaire
4. = équation algébro-différentielle

8.2 Formulation matricielle

Soit $\underline{\underline{C}}$ la matrice des contraintes. Notons N_c le nombre d'équations de contraintes, ainsi $\underline{\underline{C}}$ est de taille $N_c \times N_{\text{ddl}}$:

$$\begin{array}{rc} \text{Equation dynamique} & \left\{ \begin{array}{lll} \underline{\underline{A}}\underline{\ddot{q}} & -\underline{\underline{C}}^t\underline{\lambda} & = \underline{B} \\ \underline{\underline{C}}\underline{\ddot{q}} & & = \underline{D} \end{array} \right. \\ \text{Contrainte avec Baumgarte} & \end{array}$$

La formulation matricielle s'écrit :

$$\begin{bmatrix} \underline{\underline{A}} & \underline{\underline{C}}^t \\ \underline{\underline{C}} & \underline{\underline{0}} \end{bmatrix} \begin{Bmatrix} \underline{\ddot{q}} \\ -\underline{\lambda} \end{Bmatrix} = \begin{Bmatrix} \underline{B} \\ \underline{D} \end{Bmatrix} \tag{20}$$

Une fois mise sous forme d'état, on trouve :

$$\begin{bmatrix} \underline{\underline{I}} & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{A}} & \underline{\underline{C}}^t \\ \underline{\underline{0}} & \underline{\underline{C}} & \underline{\underline{0}} \end{bmatrix} \begin{Bmatrix} \underline{\dot{q}} \\ \underline{\ddot{q}} \\ -\underline{\lambda} \end{Bmatrix} = \begin{Bmatrix} \underline{\dot{q}} \\ \underline{B} \\ \underline{D} \end{Bmatrix} \tag{21}$$

On utilisera principalement la forme (20) dans Python, mais notons qu'il est possible de résoudre d'abord $\underline{\lambda}$ puis $\underline{\ddot{q}}$:

$$\underline{\underline{A}}\underline{\ddot{q}} - \underline{\underline{C}}^t\underline{\lambda} = \underline{B} \quad \Longrightarrow \quad \boxed{\underline{\ddot{q}} = \underline{\underline{A}}^{-1}\underline{B} + \underline{\underline{A}}^{-1}\underline{\underline{C}}^t\underline{\lambda}} \quad \Longrightarrow \quad \underline{D} = \underline{\underline{C}}\underline{\ddot{q}} = \underline{\underline{C}}\underline{\underline{A}}^{-1}\underline{B} + \underline{\underline{C}}\underline{\underline{A}}^{-1}\underline{\underline{C}}^t\underline{\lambda}$$

$$\Longrightarrow \quad \boxed{\underline{\lambda} = \left(\underline{\underline{C}}\underline{\underline{A}}^{-1}\underline{\underline{C}}^t\right)^{-1}\left(\underline{D} - \underline{\underline{C}}\underline{\underline{A}}^{-1}\underline{B}\right)}$$

Il existe donc différentes approches pour créer la fonction $\underline{\dot{y}} = \underline{f}(t, \underline{y})$ qui sera intégrée par `scipy.integrate.solve_ivp`.

8.2.1 Variante 1

On peut considérer que les multiplicateurs de Lagrange font partie du vecteur d'état, lequel devient $\underline{y}^t = \langle \underline{q}^t, \underline{\dot{q}}^t, -\int \underline{\lambda}^t \rangle$ et sa dérivée $\underline{\dot{y}}^t = \langle \underline{\dot{q}}^t, \underline{\ddot{q}}^t, -\underline{\lambda}^t \rangle$. Dans ce cas, on calcule le vecteur temporaire :

$$\underline{temp} = \begin{Bmatrix} \underline{\ddot{q}} \\ -\underline{\lambda} \end{Bmatrix} = \begin{bmatrix} \underline{\underline{A}} & \underline{\underline{C}}^t \\ \underline{\underline{C}} & \underline{\underline{0}} \end{bmatrix} \backslash \begin{Bmatrix} \underline{B} \\ \underline{D} \end{Bmatrix}$$

Pseudo-code Python 1.

```
def _odefun(self, t, y):
    temp = ...
    dy[0 : Nq] = y[Nq : 2*Nq]
    dy[Nq : 2*Nq+Nc] = temp
    return dy
```

8.2.2 Variante 2

On peut calculer les multiplicateurs de Lagrange comme variable temporaire et considérer qu'ils ne font pas partie du vecteur d'état, alors $\underline{y}^t = \langle \underline{q}^t, \underline{\dot{q}}^t \rangle$ et sa dérivée $\underline{\dot{y}}^t = \langle \underline{\dot{q}}^t, \underline{\ddot{q}}^t \rangle$. Dans ce cas :

$$\underline{\lambda} = \left(\underline{\underline{C}}\underline{\underline{A}}^{-1}\underline{\underline{C}}^t\right)^{-1}\left(\underline{D} - \underline{\underline{C}}\underline{\underline{A}}^{-1}\underline{B}\right) \qquad \text{et} \qquad \underline{\ddot{q}} = \underline{\underline{A}}^{-1}\underline{B} + \underline{\underline{A}}^{-1}\underline{\underline{C}}^t\underline{\lambda}$$

Pseudo-code Python 2.

```
def _odefun(self, t, y):
    Lambda = ...
    d2q = ...
    dy[0 : Nq] = y[Nq : 2*Nq]
    dy[Nq : 2*Nq] = d2q
    return dy
```

8.2.3 Variante 3

Les accélérations et les multiplicateurs de Lagrange sont calculés en même temps comme dans la variante 1. En revanche, le vecteur d'état ne contient pas les multiplicateurs de Lagrange, comme dans la variante 2 : $\underline{y}^t = \langle \underline{q}^t, \underline{\dot{q}}^t \rangle$ et sa dérivée vaut $\underline{\dot{y}}^t = \langle \underline{\dot{q}}^t, \underline{\ddot{q}}^t \rangle$.

Pseudo-code Python 3.

```
def _odefun(self, t, y):
    temp = ...
    dy[0 : Nq] = y[Nq : 2*Nq]
    dy[Nq : 2*Nq] = temp[0 : Nq]
    return dy
```

La variante 1 conduit éventuellement à ce que le terme $\int \underline{\lambda}^t$ tende vers l'infini. Cela pose des problèmes numériques. Les variantes 2 et 3 sont quasiment aussi précises l'une que l'autre mais la variante 3 est un peu plus rapide et plus simple à programmer.

8.2.4 Interprétation de $\underline{\lambda}$

Le terme $\left(\underline{\dot{q}}^*\right)^t \underline{\underline{C}}^t\underline{\lambda}$ est homogène à $\mathscr{A}^*$ et $\mathscr{P}^*$, c'est-à-dire à une puissance virtuelle. L'écriture $\underline{\underline{A}}\underline{\ddot{q}} = \underline{B} + \underline{\underline{C}}^t\underline{\lambda}$ prouve que le terme $\underline{\underline{C}}^t\underline{\lambda}$ est homogène à $\underline{B}$, c'est-à-dire à un effort généralisé ou plus simplement à un torseur d'effort exprimé au centre de gravité du solide.

Dans certains cas, les termes de la matrice $\underline{\underline{C}}$ ne sont pas sans dimension alors l'interprétation de $\underline{\lambda}$ est compliquée. Considérons le cas le plus fréquent où $\underline{\underline{C}}$ est sans dimension, alors $\underline{\lambda}$ est homogène à un effort ou à un couple. Lorsque deux solides sont en relation, le signe des termes de $\underline{\underline{C}}^t$ permet ensuite de savoir si $\underline{\lambda}$ est l'effort de S_A sur S_B ou l'inverse.

8.3 Quelques exemple de contraintes

8.3.1 Pivot (= Rotule en 2D)

Le point A du solide S_A est confondu avec le point B du solide S_B, la contrainte $\underline{g} = \underline{0}$ s'écrit :

$$\underline{g} = \underline{AB}^{\mathscr{R}_0} = \underline{OB}^{\mathscr{R}_0} - \underline{OA}^{\mathscr{R}_0} = \underline{0}$$

La contrainte de liaison pivot devient :

$$\begin{cases} \underline{g} = \underline{OB}^{\mathscr{R}_0} - \underline{OA}^{\mathscr{R}_0} = \underline{0} \\ \\ \dot{\underline{g}} = \dot{\underline{OB}}^{\mathscr{R}_0} - \dot{\underline{OA}}^{\mathscr{R}_0} = \underline{0} = \underline{\underline{T}}_v(S_B, B)\dot{\underline{q}}_B \ - \ \underline{\underline{T}}_v(S_A, A)\dot{\underline{q}}_A \\ \\ \ddot{\underline{g}} = \ddot{\underline{OB}}^{\mathscr{R}_0} - \ddot{\underline{OA}}^{\mathscr{R}_0} = \underline{0} = \left(\underline{\underline{T}}_v(S_B, B)\ddot{\underline{q}}_B + \underline{\underline{T}}_a(S_B, B)\dot{\underline{q}}_B\right) \\ \qquad\qquad\qquad\qquad - \ \left(\underline{\underline{T}}_v(S_A, A)\ddot{\underline{q}}_A + \underline{\underline{T}}_a(S_A, A)\dot{\underline{q}}_A\right) \end{cases}$$

On peut maintenant exprimer la contrainte sous forme matricielle :

$$\underbrace{-\underline{\underline{T}}_v(S_A, A)}_{=\underline{\underline{C}}_A}\ \ddot{\underline{q}}_A \ + \underbrace{\underline{\underline{T}}_v(S_B, B)}_{=\underline{\underline{C}}_B}\ \ddot{\underline{q}}_B \ = \ \underbrace{\underline{\underline{T}}_a(S_A, A)\ \dot{\underline{q}}_A \ - \underline{\underline{T}}_a(S_B, B)\ \dot{\underline{q}}_B}_{=\underline{D}}$$

Notons :

$$\begin{cases} \underline{\underline{C}}_A &= -\ \underline{\underline{T}}_v(S_A, A) \\ \underline{\underline{C}}_B &= \ \ \underline{\underline{T}}_v(S_B, B) \end{cases} \quad \text{et} \quad \begin{aligned} \underline{D} &= \underline{\underline{T}}_a(S_A, A)\dot{\underline{q}}_A \qquad - \underline{\underline{T}}_a(S_B, B)\dot{\underline{q}}_B \\ &= \dot{\theta}_A^2 \underline{\underline{R}}_a(\theta_A)\underline{G_A A}^{\mathscr{R}_A} \ - \dot{\theta}_B^2 \underline{\underline{R}}_a(\theta_B)\underline{G_B B}^{\mathscr{R}_B} \end{aligned}$$

Ici, la contrainte $\ddot{\underline{g}} = \underline{\underline{C}}\ddot{\underline{q}} - \underline{D} = \underline{0}$ est homogène à une accélération, donc $\underline{\underline{C}}$ est sans dimension. De plus, $\underline{\underline{C}}_A$ est négatif donc on trouve des "-1" sur les lignes de $\underline{\underline{C}}^t$ correspondant à l'accélération $\ddot{x}_A$ et $\ddot{z}_A$ et des "1" sur les lignes de $\underline{\underline{C}}^t$ correspondant à l'accélération $\ddot{x}_B$ et $\ddot{z}_B$, on en déduit que le multiplicateur de Lagrange $\underline{\lambda}$ représente l'effort de liaison du solide S_A sur le solide S_B dans le repère Galiléen.

Considérons à titre d'exemple un système multicorps composé de 3 solides. Supposons que nous cherchions à créer une liaison pivot entre le point A du solide S3 (ici égal au solide S_A) et le point B du solide S2 (ici égal au solide S_B) : $\underline{AB}^{\mathscr{R}_0} = \underline{0}$. Ainsi le solide S1 ne participerait pas à la contrainte. En pratique, l'équation (19) s'écrirait :

$$\begin{cases} \underline{\underline{C}} = \left[\underline{\underline{0}}_{2\times 3}\ ,\ \underline{\underline{C}}_B\ ,\ \underline{\underline{C}}_A\right] \\ \ddot{\underline{q}}^t = \left\langle\ \ddot{x}_1 \quad \ddot{z}_1 \quad \ddot{\theta}_1 \quad \ddot{x}_2 \quad \ddot{z}_2 \quad \ddot{\theta}_2 \quad \ddot{x}_3 \quad \ddot{z}_3 \quad \ddot{\theta}_3\ \right\rangle \end{cases}$$

Cela montre que les classes d'interactions qui implémentent des contraintes doivent connaitre les indices des solides dans le vecteur d'état global. C'était déjà une information nécessaire pour les classes d'interaction qui implémentent des lois de comportement. Ces dernières en avaient besoin pour calculer les termes du vecteur $\underline{B}$, tout comme nous en avons besoin ici pour calculer

les termes des matrices $\underline{\underline{C}}_A$, $\underline{\underline{C}}_B$ et $\underline{D}$. Mais dans le cas présent nous en auront également besoin pour assembler les matrices $\underline{\underline{C}}_A$, $\underline{\underline{C}}_B$ dans la matrice $\underline{\underline{C}}$.

En reprenant l'exemple précédent, on aurait au final :

$$\begin{cases} \underline{\underline{C}} = \left[\underline{\underline{0}}_{2\times 3} \ , \ \underline{\underline{C}}_B \ , \ \underline{\underline{C}}_A\right] \\ \underline{D}_{\text{bg}} = \underline{\underline{T}}_a(S_A, A)\dot{\underline{q}}_A - \underline{\underline{T}}_a(S_B, B)\dot{\underline{q}}_B \ - 2\xi_{\text{bg}}\omega_{\text{bg}}\left(\dot{\underline{OB}}^{\mathscr{R}_0} - \dot{\underline{OA}}^{\mathscr{R}_0}\right) \ - \omega_{\text{bg}}^2\left(\underline{OB}^{\mathscr{R}_0} - \underline{OA}^{\mathscr{R}_0}\right) \end{cases}$$

Nous allons maintenant mettre à jour le code Python :

- Les matrices $\underline{\underline{C}}$ et $\underline{D}$ sont déjà initialisées dans la méthode `Model._build_interactions()`
- La classe `Param` contiendra les paramètres de Baumgarte.
- La classe d'interaction `Hinge_2s` implémentera la liaison pivot entre les solides S_A et S_B

Code Python 19.
Modifiez la classe `Param`

```
class Param:
    def __init__(self):
        self.g = 9.81
        self.bmgt_xi = 1
        self.bmgt_w0 = 2*np.pi*10
        self.bmgt_tau = 0.1
```

Code Python 20.
A la suite du fichier Python **Multicorps2D.py**
Implémentez la classe `Hinge_2s`

```
class Hinge_2s:
   def __init__(self, label="Hinge_2s",
                tag_S=["tag_Sa", "tag_Sb"],
                tag_F=["tag_Fa", "tag_Fb"]):
       # Common properties
       self.label  = label  # Label of the interaction
       self.n_c    = 2      # Number of contraints equations
       self.tag_S  = tag_S  # Tags of the solids in interaction
       self.tag_F  = tag_F  # Tags of the frames in interaction
       # Specific properties
       # Parameters of Solid A
       self.ind_q_a = []
       self.ind_dq_a = []
       self.GP_a = []
       # Parameters of Solid B
       self.ind_q_b = []
       self.ind_dq_b = []
       self.GP_b = []

   def _build(self, solids, param):
       # Solid Sa
       s_a = solids[self.tag_S[0]]
```

```
            f_a = s_a.frames[self.tag_F[0]]
            self.ind_q_a = s_a.ind_q
            self.ind_dq_a = s_a.ind_dq
            self.GP_a = f_a.position
            # Solid Sb
            s_b = solids[self.tag_S[1]]
            f_b = s_b.frames[self.tag_F[1]]
            self.ind_q_b = s_b.ind_q
            self.ind_dq_b = s_b.ind_dq
            self.GP_b = f_b.position
            # Baumgarte parameters
            self.bmgt_2xiw0 = 2 * param.bmgt_xi * param.bmgt_w0
            self.bmgt_w02 = param.bmgt_w0**2

        def assemble(self, t, y, matrices):
            # Parameters of Solid A
            q_a = y[self.ind_q_a]
            dq_a = y[self.ind_dq_a]
            GP_a = self.GP_a
            # Parameters of Solid B
            q_b = y[self.ind_q_b]
            dq_b = y[self.ind_dq_b]
            GP_b = self.GP_b
            # Acceleration terms
            Ca = - transformation_v(q_a, GP_a)
            Cb = + transformation_v(q_b, GP_b)
            Da = transformation_a(q_a, dq_a, GP_a) @ dq_a
            Db = transformation_a(q_b, dq_b, GP_b) @ dq_b
            D = Da - Db
            # Velocity term d_AB
            d_AB = velocity(q_b, dq_b, GP_b) - velocity(q_a, dq_a, GP_a)
            # Position term AB
            AB = position(q_b, GP_b) - position(q_a, GP_a)
            # Matrix assembly
            matrices.C[np.ix_(self.ind_c, self.ind_q_a)] = Ca
            matrices.C[np.ix_(self.ind_c, self.ind_q_b)] = Cb
            matrices.D[self.ind_c] = D - self.bmgt_2xiw0 * d_AB \
                - self.bmgt_w02 * AB
            return matrices
```

Exercice 9.
Implémentez la classe `Hinge_1sFix` qui lie un solide à un point du repère galiléen.

```
class Hinge_1sFix:
    def __init__(self, label="Hinge_1sFix",
                 tag_S="tag_S", tag_F="tag_F",
                 P_Gal=np.array([1, 1])):

    ?___?
```

8.3.2 Equidistance entre A et B

$$\left\| \underline{AB}^{\mathscr{R}_0} \right\| = L \qquad \Longrightarrow \qquad \underline{g} = \sqrt{\left(\underline{AB}^{\mathscr{R}_0}\right)^t \underline{AB}^{\mathscr{R}_0}} - L = 0$$

8.3.3 Linéaire annulaire : A contraint sur l'axe $\underline{Z}_{L_B}$

$$\underline{g} = \left(\underline{AB}^{\mathscr{R}_0}\right)^t \underline{Z}_{L_B} = 0$$

8.3.4 Glissière d'axe $\underline{Z}_{L_B}$

En considérant un angle $\Theta_{constante}$ entre les solides S_A et S_B.

$$\underline{g} = \left\{ \begin{array}{c} \left(\underline{AB}^{\mathscr{R}_0}\right)^t \underline{Z}_{L_B} \\ \theta_B - \theta_A - \Theta_{constante} \end{array} \right\} = \underline{0}$$

8.3.5 Liaison rigide entre A et B

$$\underline{g} = \left\{ \begin{array}{c} \underline{AB}^{\mathscr{R}_0} \\ \theta_B - \theta_A \end{array} \right\} = \left\{ \begin{array}{c} x_B^{\mathscr{R}_0} - x_A^{\mathscr{R}_0} \\ z_B^{\mathscr{R}_0} - z_A^{\mathscr{R}_0} \\ \theta_B - \theta_A \end{array} \right\} = \underline{0}$$

8.3.6 Cinématique imposée d'un solide

Soit $\underline{f}(t)$ un vecteur arbitraire à 3 composantes qui dépendent du temps

$$\underline{g} = \underline{q} - \underline{f}(t) = \underline{0}$$

Exercice 10.
Implémentez la classe `Motion_1s` qui impose le mouvement d'un solide.

```
class Motion_1s:
    def __init__(self, label="Motion_1s",
                 tag_S="tag_S", fun=tabular()):

    ?___?
```

9 Post-traitement

9.1 Animation temporelle

Il est indispensable de pouvoir visualiser le mouvement du mécanisme pour savoir si ce dernier ce comporte comme prévu et donc si la simulation ne comporte pas de bugs : revoir le cours de Python ++.

9.2 Affichage des repères

Il est utile d'afficher la position et l'orientation des repères.

9.3 Autres grandeurs physiques

Il peut être utile de simuler ce qu'aurait mesuré un capteur sur le système multicorps :

- Accélération en un point P d'un solide S
- Effort de liaison entre 2 solides
- Effort produit par un ressort, ...

A l'issu du calcul, `scipy.integrate.solve_ivp` donne le vecteur d'état du système en fonction du temps, c'est-à-dire la position $\underline{q}$ et la vitesse $\underline{\dot{q}}$ de chaque solide. L'accélération $\underline{\ddot{q}}$ et les multiplicateurs de Lagrange $\underline{\lambda}$ doivent donc être recalculés après coup (post-traités).

Les accélérations peuvent être recalculées avec la méthode `dy = Model._odefun(t, y)` sans faire d'approximation numérique. En revanche seule la variante 1, cf. section 8.2.1 permet de calculer aussi les multiplicateurs de Lagrange et ce n'est pas cette variante que nous avons choisi d'implémenter. Il est donc nécessaire d'implémenter une méthode de post-traitement qui ressemble à `Model._odefun` mais qui retourne toutes les variables intermédiaires : positions, vitesses, accélérations, efforts de d'interation et multiplicateurs de Lagrange.

Pensez à reprojeter les multiplicateurs de Lagrange du repère Galiléen vers le repère du solide. On prend de cette manière le point de vue du solide, ce qui est bien utile si on souhaite par la suite réaliser une étude de ce solide par éléments finis, dans Abaqus par exemple :

$$\underline{F}^{\mathscr{R}_S} = \underline{\underline{R}}_{S\leftarrow 0}\underline{\lambda} = \underline{\underline{R}}^t_{0\leftarrow S}\underline{\lambda}$$

Pensez à également à reprojeter les accélérations du repère Galiléen vers le repère de liaison du solide. On prend de cette manière le point de vue d'un accéléromètre dont l'orientation dépend de celle du solide sur lequel il est fixé :

$$\underline{\ddot{q}}^{\mathscr{R}_L}_P = \underline{\underline{R}}_{L\leftarrow S}\underline{\underline{R}}_{S\leftarrow 0}\underline{\ddot{q}}^{\mathscr{R}_0}_P = \underline{\underline{R}}^t_{S\leftarrow L}\underline{\underline{R}}^t_{0\leftarrow S}\underline{\ddot{q}}^{\mathscr{R}_0}_P$$

10 Exercices

10.1 Cas d'un simple pendule

Le point P_{Gal} est un point fixe du repère Galiléen tel que $\underline{OP}^{\mathscr{R}_0} = \langle 0, 0 \rangle^t$.

Soit un solide S tel que :
- Sa masse vaut $m = 1\text{kg}$ et son inertie vaut $I_y = 10^{-2}\text{kg m}^{-2}$
- A l'instant $t = 0$, son centre de gravité est situé en $\underline{OG}^{\mathscr{R}_0} = \langle 1, 0 \rangle^t$
- Il possède un point P confondu avec P_{Gal} à l'instant $t = 0$
- Le solide S est en liaison pivot avec le galiléen de telle sorte que les points P et P_{Gal} restent confondus à tout instant.

Exercice 11.
Programmez le script principal **Main_SimplePendule** qui
- met en données le problème
- initialise le vecteur d'état
- intègre le système pour $t \in [0..5]$
- trace l'accélération du centre de gravité dans le repère du solide
- trace l'effort de liaison dans le repère du solide

10.2 Ciseaux

Soit un système multicorps composé de 4 solides tels que représentés dans la figure 6. Leur masse vaut $m = 1\text{kg}$ et leur inertie vaut $I_y = 10^{-2}\text{kg}\,\text{m}^{-2}$. Les paramètres du ressort/amortisseur sont $k = 1000$ N/m, $c = 100$ N s/m et $L_0 = 4$ m.

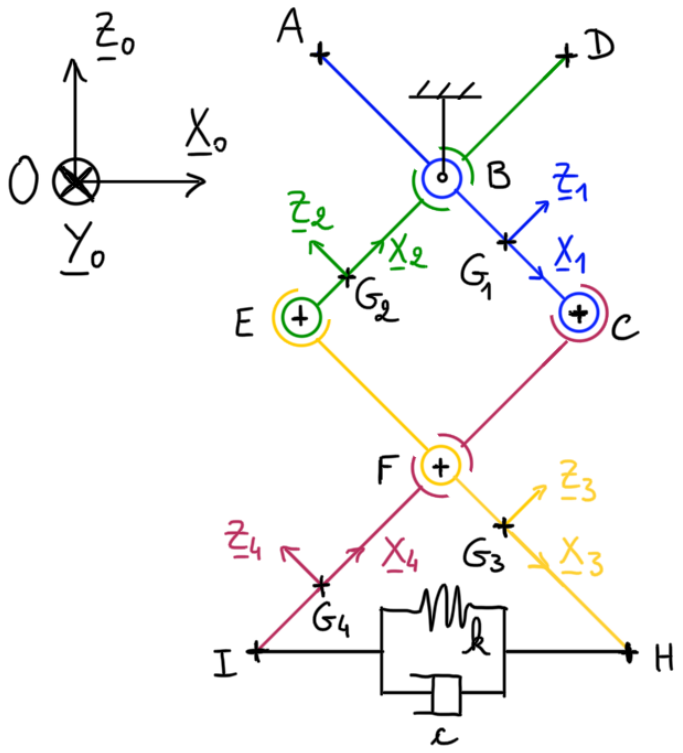

FIGURE 6 – *Position initiale des solides composant les ciseaux*

Notez que :
- le solide 1 (bleu) comporte les points A, B, G_1 et C
- le solide 2 (vert) comporte les points D, B, G_2 et E
- le solide 3 (jaune) comporte les points E, F, G_3 et H
- le solide 4 (rose) comporte les points C, F, G_4 et I

Ces solides sont liés entre eux par :
- une liaison pivot au point B entre les solides 1, 2 et le bâti
- une liaison pivot au point C entre les solides 1 et 4
- une liaison pivot au point E entre les solides 2 et 3
- une liaison pivot au point F entre les solides 3 et 4
- un ressort/amortisseur (k, c) entre les points I et H appartenant aux solides 3 et 4

A l'instant initial, les points sont placés comme suit :

points	A	B	C	D	E	F	H	I	G_1	G_2	G_3	G_4
x =	-1	0	1	1	-1	0	1	-1	0.5	-0.5	0.5	-0.5
z =	1	0	-1	1	-1	-3	-5	-5	-0.5	-0.5	-4.0	-4.0

TABLE 1 – *Position initiale des points de la structure*

Exercice 12.
Programmez le script principal **Main_Ciseaux** qui
- met en données le problème
- initialise le vecteur d'état
- intègre le système pour $t \in [0..5]$
- trace les efforts de liaison dans le repère des solides
- anime le résultats

10.3 Cas d'une chaine de vélo

Soit une chaine de longueur 1m et composée d'un nombre arbitraire de maillons.

- Chaque maillon a une masse $m = 0.01\text{kg}$ et une inertie $I_y = 10^{-9}\text{kg}\,\text{m}^{-2}$.
- Chaque maillon est composé d'un point P1 et d'un point P2. Le centre de gravité G du maillon est à mi-distance de P1 et P2
- Le point A est un point fixe du repère Galiléen tel que $\underline{OA}^{\mathscr{R}_0} = \langle 2, 5\rangle^t$
- Le P2 du solide k est en liaison pivot avec le point P1 du solide k+1

La figure 7 montre l'état initial de la chaine.

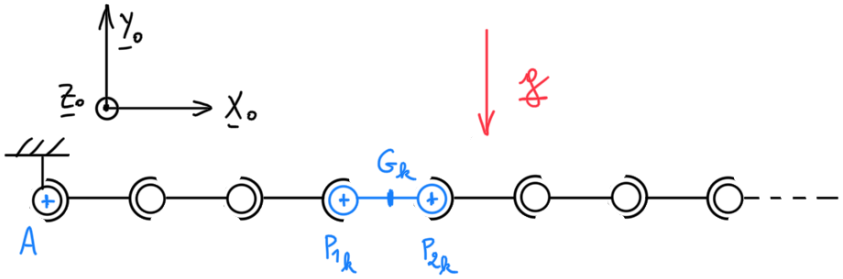

FIGURE 7 – *Position initiale de la chaine de vélo*

Exercice 13.
Simuler la chaine de vélo sur une durée $t \in [0, 1.5]$ s.

Animez la chaine de vélo.